VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zpracování dat z Wikipedie

Wikipedia Data Processing

2019 Martin Mikula

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Mikula**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Zpracování dat s Wikipedie
Wikipedia Data Processing

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce bude využít parser dat z Wikipedie a exportovaná data analyzovat pomocí metod pro analýzu sítí nebo textových dat.

Práce bude obsahovat:
1. Přehled algoritmů pro analýzy dat z Wikipedie.
2. Popis vybraných algoritmů.
3. Návrh implementace a implementaci popsaných metod.
4. Porovnání implementovaných metod.

Seznam doporučené odborné literatury:

[1] Barabási, Albert-László. "Network science." Cambridge University Press; 978-1107076266. 2016
[2] Crochemore, Maxime, and Wojciech Rytter. Jewels of stringology: text algorithms. World Scientific, 2003.
[3] Gusfield, Dan. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press, 1997.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: *26. dubna 2019*

.....
podpis studenta

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli. Děkuji mému vedoucímu diplomové práce, docentu Janu Platošovi, který mi udával správný směr a ochotně odpovídal na moje dotazy. Také chci poděkovat svým blízkým, kteří mě po celou dobu podporovali. Velké poděkování také patří mému nadřízenému v zaměstnání, panu Antonínu Vaněčkovi, který vyhověl každé mé žádosti o osobní volno. A bez toho by tato práce nikdy nevznikla.

Abstrakt

Cílem této diplomové práce je zdokumentovat možnosti zpracování dat Wikipedie. V první části popisuje způsob, jak tato data získat, zpracovat a uložit pro další analýzu. Přitom je na databázi nahlíženo jako na síť a zaměření je na provázání stránek mezi sebou pomocí odkazů. Samotná analýza probíhá v prostředí Python. Práce popisuje, jak vytvořit graf a jak nad tímto grafem spočítat základní vlastnosti a metriky. Dále je zdokumentován postup hledání komunit v grafu včetně vlastní implementace algoritmu Label Propagation. Prezentovány jsou výsledky jednotlivých kroků.

Klíčová slova: Wikipedie, analýza dat, zpracování dat, C, Python, síť, graf, CSR, NetworkX, Gephi, word cloud

Abstract

Goal of this master thesis is to describe options of how to process data from Wikipedia. First part is about how to get the data, process them and save for further analysis. The database is viewed as a network, so it's focused on pages and their connections through links. The analysis is made in Python environment. Thesis describes how to create a graph and how to calculate his basic properties an metrices. It further documents the procedure of finding the communities, including custom implementation of Label Propagation algorithm. Presented are results of each step.

Key Words: Wikipedia, data analysis, data processing, C, Python, network, graph, CSR, NetworkX, word cloud

Obsah

Seznam použitých zkratek a symbolů

wiki	–	Typ Webové stránky umožňující uživatelům přidávat obsah.
Wikipedia	–	Webová encyklopedie tvořená svobodným obsahem. Využívá software MediaWiki.
MediaWiki	–	Svobodný software umožňující vytvářet systémy wiki. Je využíván jako jádro systému Wikipedie a spol..
Wikimedia	–	Nadace spravující projekty Wikipedia, MediaWiki a další.
CSV	–	Comma Sepparated Values. Formát uložení dat v textové podobě. Jednotlivé hodnoty jsou odděleny znakem, obvykle čárkou, nebo středníkem.
LPA	–	Label Propagation Alrorythm je metoda pro hledání komunit v grafu.
Dictionary	–	V českém jazyce slovník. V této práci ovšem označení pro datovou strukturu jazyka C#. Obdoba této třídy v JAVA je HashTable.
CSR	–	Zkratka z anglického Compressed Sparse Row. Jedná se o úsporný způsob uložení řídkých matic.
Dump	–	V souvislostí s databází je dump kopie dat, nebo struktury databáze. Její účel může být například záloha, nebo sdílení obsahu.

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Wikipedie je online encyklopedie, která vzniká přispíváním mnoha dobrovolníků po celém světě. Jedná se pravděpodobně o největší veřejně dostupnou databází vědomostí na internetu. Někdo jí považuje za samozřejmost, někdo jí označuje jako kolektivní vědomí a mnoho z nás jako zdroj pravdy. Ať už jsou tyto výroky pravdivé nebo ne, máme k dispozici ohromné množství informací, které si na Wikipedii můžeme přečíst. Chybí ale možnost, jak jednoduše tato data zpracovat pomocí počítačových programů.

V této práci jsem se proto zaměřil na způsob, jak z databáze Wikipedie získat data ve strojově čitelné podobě a jak je zpracovat tak, abychom z nich mohli čerpat co nejvíce informací. Později jsem se zaměřil na část těchto informací a na data jsem pohlížel jako na síť stránek propojených přes odkazy. První část této práce je popisem tohoto procesu. Pro tyto účely jsem vytvořil sadu programů. Jejich úkolem je z této sítě sestavit graf a uložit ho do souboru tak, aby byl vhodný pro další použití a analýzu. V druhé části z tohoto souboru vycházím a v prostředí Python graf postupně analyzuji. Popisuji zde knihovny, které jsou k tomu vhodné a ukazuji, jak s jejich pomocí získat o grafu základní informace. Následuje hledání komunit a vysvětlení problémů, které jsem v souvislosti s tímto úkolem měl. V rámci řešení jsem vytvořil vlastní implementaci algoritmu pro hledání komunit. Jeho výsledky jsem prezentoval vykreslením celého grafu a vybrané komunity jsou vyobrazeny metodou word cloud. V průběhu celé této práce se snažím upozorňovat na komplikace, na které jsem v průběhu vývoje narazil. Především pak na ty, které plynou z velkého množství dat a jejichž řešení pro mě bylo skutečnou výzvou.

2 Popis zdrojových dat

Tato práce se věnuje zpracování dat z Wikipedie. Tato encyklopedie je známá jako největší open-source projekt lidstva, není tedy překvapení, že celá databáze je dostupná na internetu, především v podobě veřejně dostupných webových serverů.

Systém serverů Wikipedie je nastaven tak, aby v pravidelných intervalech prováděl kompletní zálohy všech informací, které uživatelé vytvořili. Z těch vytvoří takzvaný Dump, tedy v podstatě kopii celé databáze. Součástí procesu je následné nasdílení na několik veřejných světových serverů, které obvykle patří vysokým školám. Z těchto uložišť, zvaných Mirror, si jej může pro své potřeby kdokoli stáhnout.

Protože množství informací je opravdu velké, není celá tato záloha v jednom souboru. Naopak je rozdělena na mnoho částí. A základním rozdělením jsou jazyky, nebo jejich verze. Dále se pak zvlášť ukládají samotné články a s nimi související informace, jako průběžné změny v čase, informace o uživatelích, obrázky a další.

Pro tuto práci jsem zvolil databázi zvanou Simple English. Jedná se o databázi v anglickém jazyce psanou tak, aby byla srozumitelná i těm, kteří ovládají řeč na nižší úrovni. Databáze vzniká paralelně s hlavní anglickou databází a její obsah je vůči ní omezený. Pro mou práci je tento výběr ideální i pro menší množství stránek, všechny použité procesy by ovšem byly stejné i pro jakoukoli jinou databázi.

2.1 Formát zdrojových dat

Soubor, který se stáhne z úložiště, je komprimovaný balíček ve formátu bzip2 o velikosti 137MB. Po jeho rozbalení získáme jediný XML soubor, který je velký 572 MB. V tomto souboru jsou pak jednotlivé stránky. Každá stránka se skládá z několika metainformací. Zpravidla jsou to tyto: titulek, ID stránky, namespace, autor, čas poslední změny, formát textu, nebo kontrolní součet.

V neposlední řadě je to samotný text. Text stránky je v XML jako jedna hodnota a je zapsaný ve formátu Wikitext.

2.2 Wikitext

Wikitext je podobně jako třeba xml značkovací jazyk. Jde o zdrojový text, který je určen pro strojové čtení. Tento text je odlehčený tak, aby co nejvíce připomínal prostý psaný text. Důvodem zjednodušení je, aby se uživateli psaly články co možná nejsnadněji.

Pokud bychom chtěli zapsat jednoduchou větu, ve wikitextu by neobsahovala žádné speciální řídící znaky. Příkladem takových znaků může být odkaz na jinou stránku. Odkaz vytvoříme použitím dvojitých hranatých závorek. Jiný příklad může být třeba vložení textu mezi =. To vytvoří nadpis první úrovně. Dvě a více rovnítek jsou další úrovně nadpisu.

Obvykle se tento formát používá právě ve spojení s wiki stránkami, respektive se softwarem MediaWiki, který dokáže převádět Wikitext na HTML a zpět.

3 Zpracování dat

3.1 Uložení dat na disku

Už při prvním otevření datového souboru jsem narazil na první nepříjemnost. Soubor ve formátu XML je velký 570MB. Přestože pro dnešní počítače by neměl být problém tento soubor do paměti nahrát, často si software s tímto úkolem neporadí. Některé programy rovnou vypíší chybovou hlášku o tom, že je soubor příliš velký. Jiné se soubor pokusí otevřít, ale po čase zhavarují. Najdou se ovšem i takové, které tento problém zvládnou a soubor otevřou.

Takovým programem je třeba Visual Studio Code, který také rovnou ohlásí, že je soubor příliš veliký a že pro něj vypne pokročilé indexování. Při vyhledávání řetězce v souboru poté vypíše pouze prvních 99.999 výsledků. I přes toto omezení je ale soubor velký a nepřehledný, a tak jsem pro práci s daty navrhl jednoduchou úpravu.

Jednotlivé stránky jsem se rozhodl uložit do samostatného souboru. Pro tyto účely jsem napsal jednoduchý program v c#, který xml soubor načetl a jednotlivé části uložil do souboru. Název souboru je index, tedy odpovídá pozici ve zdrojovém souboru. Spolu s těmito soubory jsem si také uložil txt soubor, kde je prostý odřádkovaný seznam jednotlivých souborů. Tento index pak zjednodušuje psaní programů, které data načítají ve smyčce.

Toto celé jsem uložil do podsložky s názvem simplewiki_XML. Obdobnou strukturu souborů jsem zachoval také u dalších kroků zpracování.

3.1.1 Titulek jako ID

Přestože každý článek na Wikipedii má svoje ID, pro navigaci mezi stránkami se ID nepoužívá. Místo toho se pro navigaci používá titulek stránky. Tedy to, co je v hlavním nadpisu. Nejspíš je to kvůli tomu, aby byly odkazy čitelnější a měly lepší vypovídající hodnotu. Přináší to s sebou ale jeden nepříjemný jev.

Je potřeba vyřešit, že mnoho znaků vyskytujících se v titulku stránky nelze použít v URL adrese. První takový je například mezera, která se vyskytuje v názvu většiny stránek. Mezeru nahradíme podtržítkem. Pokud vyhledáme článek o Evropské unii, dostaneme se na stránku /wiki/Evproská_unie. Podobně jsou řešena také další omezení, takže třeba článek o jazyku C# se nachází na drese /C_Sharp.

Když jsem jednotlivé stránky ukládal na disk, pokusil jsem se využít těchto pravidel a poměrně jednoduše se dá získat stejný tvar, jako je v URL adrese. Ovšem restrikce, které jsou v URL, nejsou stejné, jaké má systém Windows v souborovém systému. Příkladem může být třeba uvozovka nebo hvězdička. Takové znaky pak při zpracování buď vyvolávaly chyby, nebo docházelo k neočekávanému chování.

Proto jsem pro uložení souborů na disku zvolil pouze čísla indexu. Díky indexu lze v kterékoli fázi projektu dohledat příslušný název článku.

3.2 Parsoid

Existuje mnoho projektů, které se zabývají parsováním jazyka wikitext. Samotná asociace Mediawiki se snaží udržovat jejich seznam, který najdete na adrese https://www.mediawiki.org/wiki/Alternative_p Ovšem až na jednu výjimku, jedná se povětšinou o snahu jednotlivce. Taková řešení mají vždy nějaká omezení. Nejčastější a zároveň také nejzávažnější nedostatek je nekompletní pokrytí všeho, co syntaxe wikitext v různých verzích a variantách umožňuje.

Naštěstí existuje jeden projekt, který je vytvářen samotnou organizací Wikimedia. Jeho jméno je Parsoid a nabízí kompletní implementaci všeho, co se na Wikipedii může nacházet, a to dokonce v libovolné verzi. Jeho vývoj probíhá v PHP a jako vše okolo Wikipedie, i tento program je open-source.

Parsoid je možné stáhnout a spustit na svém počítači. V případě Wikipedie jeho spuštění nemá valný výsledek. Spletité šablony Wikipedie totiž mají mnoho závislostí a nastavit server, aby fungoval správně, by bylo velmi náročné. Naštěstí ale Parsoid existuje také jako RESTBase API. Pošleme-li tomuto API wikitext, dostaneme jako výsledek zpracovaný HTML. Lze to i opačným směrem, ale v tuto chvíli nám jde o získání HTML, se kterým se bude dobře pracovat.

K tomuto API je poskytovaná také knihovna, která zjednodušuje volání vzdálených metod. Tato knihovna je napsaná v JavaScript pro prostředí node.js. Nicméně moje práce byla započata v prostředí .NET Frameworku v jazyce C#. Využil jsem balíček NodeServices pro prostředí ASP.NET Core, který umožňuje spouštět javascriptový kód ze C#. V tuto chvíli tedy pro každý xml soubor, a tedy pro každou stránku databáze, voláme API Parsoid a jako odpověď dostáváme HTML string. Výsledek uložíme na disk do podobné struktury jako v případě XML souboru, tedy do adresáře simplewiki_HTML, kde jsou umístěny jednotlivé soubory s pořadovým číslem v názvu a k nim jeden index.txt se seznamem souborů. K tomuto datasetu jsem přidal ještě jeden index, který pro každé číslo (odpovídající názvu dokumentu) uvádí titulek dané stránky. Tento index je důležitý pro další zpracování a především pro orientaci ve výsledcích následujících kroků.

Nevýhodou tohoto řešení je značná časová náročnost. Podle času vzniku souborů se dá spočítat, jak dlouho tento program běžel. Tento čas byl téměř 40 hodin. Tato doba byla při vývoji velkou komplikací. Povětšinou jsem tedy pracoval jen s částí dat a až po odladění všech problémů spustil program nad celou sadou. Ovšem ve chvíli, kdy se mi podařilo takto všechny stránky jednou zpracovat, nebylo už potřeba spouštět program znovu. S tímto výsledkem jsem se spokojil, protože šlo o jednorázovou operaci.

3.3 Vytvoření grafu

V této fázi, když máme všechny stránky převedeny do HTML dokumentu, můžeme pohodlně získat informace které jsme hledali. Stránky ve Wikipedii mají na první pohled charakter sítě, protože obvykle je mezi stránkami velké množství odkazů, takzvaných hyperlinků. Ty umožnují

čtenáři při čtení tématu klikat na některá slova či pojmy a dostudovat si tak širší souvislosti. Na základě tohoto můžeme říct, že stránky spolu souvisí, když na sebe navzájem odkazují.

Samotné vytvoření grafu vypadá následovně. Začneme definováním vrcholů grafu. To není problém, protože v předchozím kroku jsme si vytvořili seznam všech stránek. Každá má své číslo a také titulek. Číslo tedy použijeme jako ID daného vrcholu. Titulek je pro nás důležitý, protože systém wiki jej používá pro navigaci. Jakmile máme vrcholy grafu, propojíme je hranami. Abychom využili potenciál našich dat, bude graf vážený a orientovaný. Každá hrana tedy reprezentuje odkaz z jedné stránky na druhou. Pokud se vyskytne odkaz více než jednou, vyjádříme to váhou hrany. Váha hrany je tedy celé číslo 1 a vyšší. Protože je graf orientovaný, rozlišujeme, zda odkaz vede ze stránky A do stránky B, nebo naopak z B do A.

Pro tento proces jsem napsal program, opět v jazyce C#. Ten si nejprve načte soubor s titulky stránek a vytvoří z ní kolekci typu Dictionary<string, int>. To proto, abychom mohli efektivně dohledat ID stránky podle jejího titulku. Poté otevírá jednotlivé HTML stránky a za pomoci balíčku HTMLAgilityPack v dokumentu vyhledá všechny odkazy, které vedou na jinou stránku v rámci databáze. To lze poznat na základě jednoho atributu každého odkazu. Pro každý odkaz vyhledá jeho ID v tabulce titulků a vytvoří objekt reprezentující hranu z prohledávané stránky do jiné. Pokud taková již existuje, znamená to, že se odkaz opakuje a inkrementuje váhu hrany. Takto získáme kolekci hran a náš graf je kompletní.

3.3.1 Způsob uložení v paměti

Pro další práci s tímto grafem bylo potřeba navrhnout vhodnou objektovou reprezentaci v paměti programu. Vycházel jsem z charakteru dat, protože předpokládáme, že graf bude velice řídký. Účelem bylo, aby se struktura dobře plnila a dále se s ní dobře pracovalo.

Výsledkem je objekt GraphModel, který obsahuje dvě kolekce. První je dříve zmiňovaná Dictionary s titulky a jejich ID. V grafu je představují vrcholy. Druhá kolekce obsahuje hrany, tedy objekty typu Link, které nesou ID stránky, ze které vedou, ID, na které vedou, a váhu. Neukládáme vztahy mezi každou dvojicí vrcholů, ale pouze ty, které mezi sebou mají nějakou hranu. Tím ušetříme značné množství paměti. Navíc hrany nejsou uloženy v jednoduchém Listu, ale v dvojité Dictionary. Její typ je tedy Dictionary<int, Dictionary<int, Link». Takto se dá velice rychle vyhledávat a také pohodlně procházet celou kolekci.

3.4 Export

Zvolený způsob uložení grafu nám udává interface pro další důležitou část programu, kterou je uložení grafu na disk, do souboru. V programu jsem napsal několik metod, které umožňují uložení grafu do různých formátů. Některé z nich pak také obsahují metodu pro opětovné načtení souboru a znovu vytvoření původního grafu.

Takto navržený program funguje také jako převodník mezi různými formáty grafu, byť s omezeným počtem formátů.

3.4.1 GEXF

Jako hlavní formát pro export/import grafu jsem zvolil formát GEXF. Jedná se o XML soubor, který využívá podobnou strukturu, jakou má náš model. V implemetaci jsem si pomohl využitím knihovny FPar.GEFX, která jen kopíruje strukturu XML. Definuje sadu serializovatelných objektů, a to umožňuje pro načtení a uložení GEXF souboru využít standardní serializaci XML-Serializer, která je součástí .NET Frameworku. Doplníme už pouze metody, které z GEXF grafu udělají náš model a naopak.

Výstup této metody je standardní soubor GEXF, který lze otevřít v jiných programech. Příkladem takového programu je software Gephi, který podporuje několik standardních formátů a GEXF je jeden z nich. Nevýhodou tohoto formátu je, že je lidově řečeno "upovídaný". Takto uložený graf tak v našem případě zabírá na disku prostor 278MB.

3.4.2 JSON

Během vývoje jsem vytvořil také implementaci pro serializaci do formátu JSON. Jeho určení bylo pro použití v javascriptu. Tento formát ale v rámci diplomové práce nebyl nijak využit. Pro prezentaci grafu v jakékoli javascriptové knihovně je totiž graf příliš velký. Po několika pokusech jsem tuto cestu uzavřel.

3.4.3 Matrix CSV export

Další formát, do kterého je možné takový graf uložit, je tabulka. Uvážíme-li tabulku o rozměrech n*n, kde n je počet vrcholů, tak pro každou dvojici vrcholů existují právě dvě buňky v tabulce. Tedy právě tolik, kolik může v našem grafu existovat hran mezi dvěma vrcholy. Jedna buňka pro každý směr hrany. Nevýhodou je ovšem to, že tabulka obsahuje nejen hrany, které v grafu jsou, ale také velké množství nul (0), které reprezentují hrany, které v grafu nejsou.

Tento problém se možná na první pohled nejeví tak vážně. Spočítejme ale, jak by taková tabulka vypadala při uložení v paměti. Graf, se kterým v tuto chvíli pracujeme, má něco málo přes 180 vrcholů. Tabulka tedy musí mít rozměr 180,000*180.000. Nejmenší datový typ pro uložení váhy, tedy hodnoty každé buňky, je jeden byte, pokud budeme předpokládat, že nám stačí rozpětí hodnot 0-255. Pronásobením těchto hodnot dojdeme k číslu 241,3 GB. To je nepřiměřeně hodně, zvláště když to porovnáme s velikostí původního vstupního souboru. Navíc běžný počítač tolik paměti nemá, a tak musíme soubor zapisovat postupně, což struktura našeho modelu umožňuje.

Pokud bychom tabulku uložili do souboru CSV, musíme navíc připočíst oddělovací znak. Všechny znaky, které chceme ukládat, pochází ze základní tabulky ASCII, a tak pro uložení každého znaku potřebuje pouze jeden byte. Budeme předpokládat, že hodnoty váhy budou menší než 10, tedy budou převážně jednociferné. I přesto se ale dostaneme na nejméně dvojnásobek toho, kolik zabírala tabulka v paměti. Tedy asi 482,8GB.

Doposud je navržený postup s běžným počítačem reálný. Ovšem pro použití s takto velkým grafem nejde o praktické řešení. Program tedy tuto metodu implementuje, ovšem hodí se jen pro menší grafy.

3.4.4 CSR matrix

Při hledání ideální metody pro uložení rozměrného grafu do souboru jsem nakonec vybral formát CSR matice. Ten se výborně hodí pro uložení velmi řídké tabulky, protože ukládá pouze informace, které jsou nutné k vybudování celé tabulky.

Formát CSR se skládá ze tří seznamů. Seznam první obsahuje všechny nenulové hodnoty. Druhý seznam obsahuje indexy vedoucí do pole prvního. Tyto indexy určují, na které pozici začínají nové řádky. Třetí pole pak pro každou hodnotu udává pozici na řádku. Obdobná metoda je také CSC, která tabulku ukládá po sloupečcích. Více o tomto tématu píšu v kapitole [**??**]

Při implementaci této metody jsem použil knihovnu NMath. Ta nabízí vlastní implementaci pro vytváření sparse matrix. Umožňuje snadnou práci s maticí. Nabízí operace jako násobení matic, ale především přístup k jednotlivým buňkám, a to i pro zápis. Do tabulky se zapisuje stejně jako do klasického dvourozměrného pole.

Knihovna má ovšem jeden zásadní nedostatek, který se mi nepodařilo vyřešit. Přesto, že podle standardu CSR lze zapsat do tabulky prázdné řádky, knihovna NMath si s nimi neporadí a vyhazuje výjimku. Pravděpodobně jednoduché řešení by bylo najít jinou knihovnu, která tuto chybu nemá nebo si napsat vlastní implementaci. Zvolil jsem jinou variantu.

Zamyslel jsem se nad tím, proč mám v tabulce prázdné řádky. Usoudil jsem, že pokud je řádek prázdný, nemá pro nás žádnou hodnotu a můžeme ho tedy odstranit. Nemůžeme ovšem odstranit řádek uprostřed tabulky, protože by tím pádem nesedělo pořadí řádků a sloupců. Prázdné řádky jsem proto přesunul na konec tabulky. Jsou-li prázdné řádky na konci, můžeme je z tabulky vynechat. Získáme tím asymetrickou tabulku, která je menší než původní.

Pro takový přesun jsem napsal metodu, která prohledává tabulku dvěma indexy. Jeden index čte řádky shora a hledá prázdné řádky. Druhý index jde zespodu a hledá neprázdné. Když oba indexy najdou co hledají, řádky prohodí. Ale pozor, prohodit musíme také příslušné sloupečky. Takto procházíme tabulku, dokud se indexy nepotkají.

Knihovna NMath mě v tuto chvíli už podruhé zklamala, protože při tomto algoritmu se projevilo, že její implementace je velmi pomalá. Při složitosti $O(n^2)$ pro čtení a $O(n * x)$ pro zápis (x.. počet prázdných řádků) by zabral průchod tabulkou mnoho hodin. Po optimalizacích se program provedl přibližně za 3 hodiny.

Další nepříjemné překvapení mělo nastat při serializaci výsledku do stringu. Knihovna NMath má vestavěnou metodu, která z tabulky vytvoří CSV. Nejspíše opět z důvodů špatné optimalizace jsem tuto metodu nemohl použít. Po krátkém snažení programu dojde paměť a zhavaruje. NMath tedy několikrát nenaplnil moje očekávání a neoznačil bych knihovnu jako dobrou volbu.

Serializace do CSV jsem sice musel dopsat sám, ale z NMath jsem k tomu dostal veškeré nutné informace. Tedy tři seznamy, které jsem pouze metodou Join() spojil do jednoho řádku a

vypsal do souboru. Výsledný soubor má navíc jeden řádek, kde jsou uložené ID všech stránek. Soubor má tedy 4 řádky. S prostorovou optimalizací jsem velmi spokojený. Takto zapsaná data mají na disku velikost pouze 34,4 MB. Tedy asi 12 % toho, kolik zabíral formát GEXF.

4 Základní statistiky

Když máme sestavený kompletní graf, můžeme začít s analýzou toho, jak tento graf vlastně vypadá a jaké jsou jeho základní vlastnosti. Začneme množstvím vrcholů a hran v Tabulce [1]. Zde je zajímavý rozdíl mezi počtem stránek a vrcholů v grafu. To je dáno tím, že při tvorbě grafu jsme vynechali takové stránky, které neměly žádný odkaz, ani na ně nebylo odkazováno. Jedná se totiž převážně o stránky bez obsahu, které fungují jako Aliasy a přesměrují uživatele jinam.

Tabulka 1: Základní statistiky - počet hran a vrcholů

Vlastnost	Hodnota
Počet stránek	233,593
Počet vrcholů grafu	181,975
Počet hran	4,125,350

4.1 R-lang

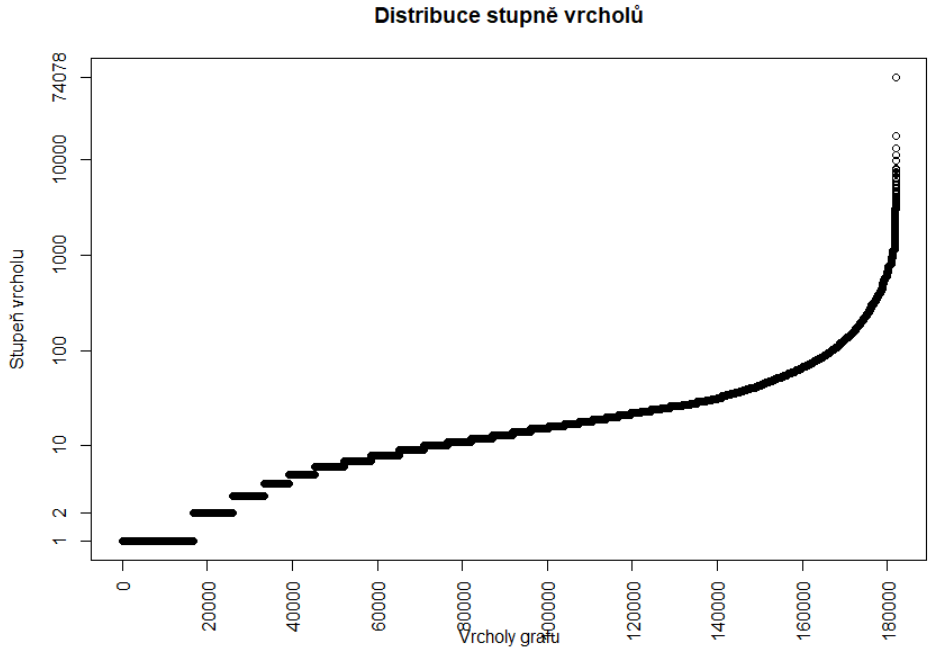
Pro další analýzu jsem některá data vyexportoval zvlášť do CSV souborů, abych tato data mohl otevřít v programu RStudio a zpracovat jazykem R. Soubory jsou dva. První obsahuje seznam všech hran a jejich vah. Druhý je seznam všech vrcholů a jejich stupeň. Hodnoty z obou souborů jsme zanesl do grafu.

4.1.1 Zhodnocení grafů

Z grafu [1] si můžeme udělat představu o vrcholech a jejich stupni. Vidíme, že největší počet vrcholů má stupeň 1, tedy jsou odkazovány jen jednou, nebo obsahují jen jeden odkaz. Dále je početná skupina se stupněm 2 a 3, a čím vyšší stupeň, tím je takových vrcholů méně. Z grafu lze poznat, že zhruba polovina hran má stupeň menší než 12. Stupeň menší než 100 má zhruba 90 % vrcholů a téměř všechny vrcholy mají stupeň menší než 10 000. Z tabulky už víme, že více než nebo takřka deset tisíc má jen 5 vrcholů. Podobně jako náš odhad hovoří také spočítaný medián, který má hodnotu 13. Průměrný stupeň vrcholu je 45.

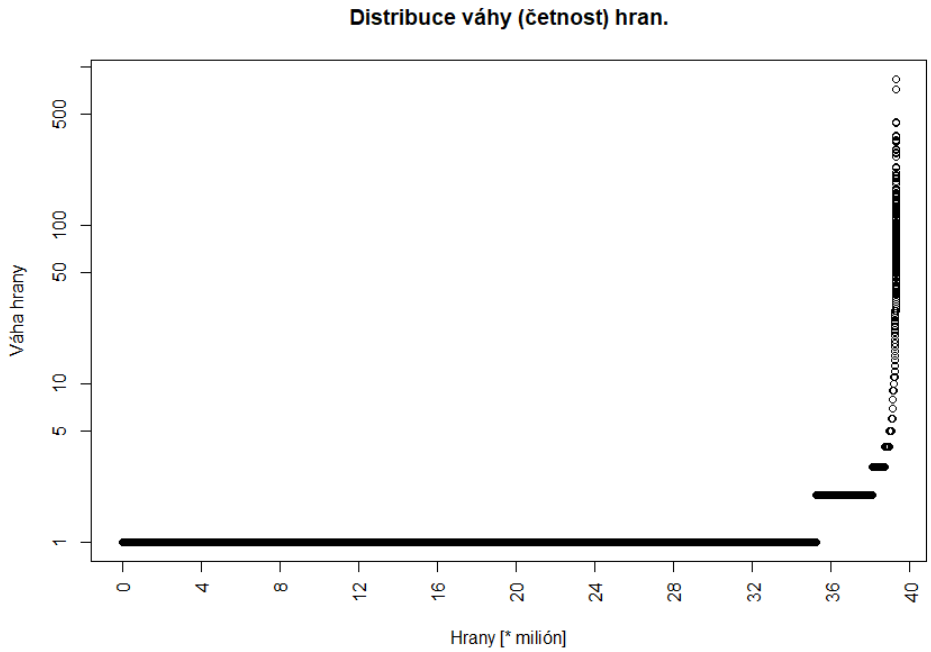
Z obrázku [2] získáme představu o distribuci váhy mezi hranami. Vidíme, jak často se mezi vrcholy vyskytuje hrana více než jednou. Je patrné, že z více než 41 miliónu hran má většina, přibližně 35 miliónu hran, váhu 1. S vahou 2 a 3 je to naprostá většina hran a vyšší stupeň má jen malé procento.

Medián nám svou hodnotou 1 mnoho nevyzradí. Zajímavý je až 99% kvantil, jehož hodnota je 4. To znamená, že 99 procent hran má váhu 4 a méně. Posuneme-li hranici ještě dál, zjistíme že 99.9% má váhu menší než 14. Průměrná hodnota je 1.2, dle očekávání velice blízká 1.



Obrázek 1: Stupně vrcholů

Stránky v naší síti jsou tedy jen velmi zřídka propojeny vetším množstvím odkazů. Většina má ale více sousedů.



Obrázek 2: Distribuce vah hran.

5 Network sampling

V kapitole o exportu grafu do souboru jsem popisoval, že zvolený formát GEXF je pro grafy standardem a lze jej tedy otevřít v programech určených pro analýzu grafu. Mezi nejznámější programy pro práci s grafy a sítěmi patří bezesporu Gephi. Jedná se o open-source software určený pro zkoumání a vykreslování sítí. Napsaný je v Javě a je tedy multiplatformní.

Při pokusu o import dat v tomto programu počítači tak říkajíc dojde dech. Už po otevření souboru dojde procesu paměť a program zhavaruje. Po pár optimalizacích v konfiguraci se soubor otevře, ale i tak je s paměti na hraně a při jakékoli operaci opět přestane pracovat.

Abych tedy vůbec mohl program využít, rozhodl jsem se pracovat pouze se vzorkem. Do mého současného řešení tak přibyl další program, který prováděl sampling grafu. Implementovány byly dvě metody. Pro obě metody platí stejný interface a jsou napsané tak, že na vstupu i na výstupu je graf ve formátu GEXF. V dalších částech tohoto dokumentu popisuji, že se mi nakonec podařilo v programu Gephi načíst větší část grafu. Program pro sampling tak zůstal bez využití v rámci této diplomové práce.

5.1 Random edge selection

Jednodušší ze dvou variant je náhodný výběr hran. Program náhodně vybírá ze všech hran z grafu a pro každou takovou do nového grafu (vzorku) přidá oba vrcholy. To se opakuje, dokud vzorek není dostatečně velký. Tedy dokud nemá dostatečný počet hran nebo vrcholů v závislosti na parametrech. Implementace není příliš pokročilá. V tomto algoritmu šlo především o vyzkoušení mechanizmu. Nejsou brány v potaz váhy hran. Pokud by váhy nebyly ignorovány, hrany s vyšším vrcholem by měly větší šanci být vybrány a výsledný vzorek by lépe reprezentoval vstupní graf. Ještě lepšího výsledku se ale dá dosáhnout použití následující metody. Nedošlo tedy na optimalizace výkonu a v případě vzorku, který se velikostí blíží původnímu grafu nastává situace, že do výsledku přibývají nové hrany jen velmi pomalu. Algoritmus v takové situaci může běžet velmi dlouho.

5.2 Random walk

Random Walk, v češtině Náhodná procházka, je metoda vzorkování. Princip spočívá v tom, že po grafu pohybujeme ukazatelem, který vždy ukazuje na některý vrchol. Počáteční vrchol můžeme zvolit náhodně, nebo může být dán parametrem. V tomto programu je počátek vždy náhodný. Z počátečního vrcholu vychází ukazatel na procházku a vybírá si náhodný směr, kam půjde. Jeho možnosti jsou všechny sousedící vrcholy. Toto se opakuje v cyklu a cesta, kterou prošel, je výsledný vzorek. Ovšem takto by mohly vznikat zvláštní "podlouhlé"grafy, kde hustota grafu je nízká a počet hran se shora blíží počtu vrcholů. Nebo by se mohla procházka zacyklit v nějakém podgrafu v rámci jedné menší komponenty a nikdy by nedosáhla k cílové velikosti. A proto přidáme do programu ještě pravděpodobnost C. Ta má obvykle hodnotu 0,1 až 0,15.

Její smysl je takový, že s pravděpodobnosti C se před následujícím krokem procházka ukončí a začne nová. Nakonec ještě zvolíme, zda má nová procházka začínat stále ze stejného počátečního vrcholu, nebo se má vybrat vždy nový náhodný vrchol.

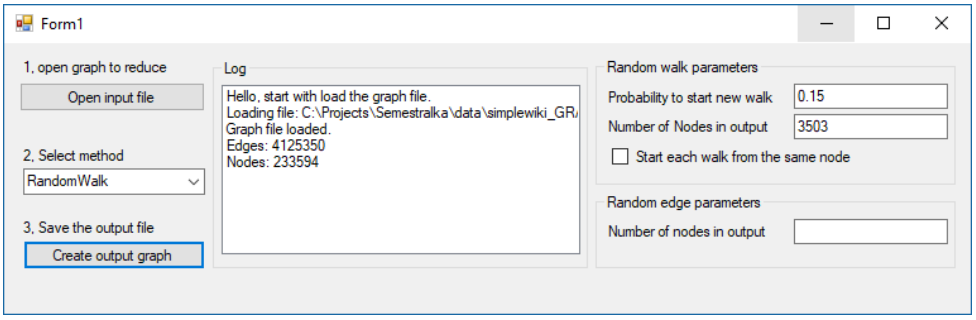
Implementace, kterou jsem napsal je zcela funkční, ovšem nedá se říct, že je kompletní. V době, kdy jsem tuto metodu psal, byl můj graf neorientovaný a neválený. Tyto vlastnosti tak algoritmus ignoruje. V případě směru hrany by bylo vhodné parametrizovat, zda může procházka procházet hranou v obou směrech, nebo má zakázáno vejít do protisměru. Váha hrany by pak měla být vzaty v potaz ve chvíli, kdy se program rozhoduje, kam směřuje jeho další krok. Hrany s vyšší vahou by měly mít větší pravděpodobnost. To se obvykle implementuje tak, že se do výběru zahrnou vícekrát.

5.3 Grafické rozhraní

Stejně jako předchozí programy, i tento je napsaný pro příkazovou řádku. Protože jsem ale vzorků dělal více a hledal ideální nastavení parametrů, vytvořil jsem pro tento program také uživatelské rozhraní.

Práce s aplikací začíná otevřením vstupního souboru s grafem. Ten je načten do paměti programu a uživatel se dozví základní informace o počtu hran a počtu vrcholů. To pomáhá při určení velikosti výstupního vzorku. Automaticky se přednastaví doporučená hodnota, nejméně 15 %, aby vzorek dostatečně odpovídal originálu. Uživatel má nyní možnost ještě parametry upravit. Nakonec nechá vytvořit vzorek a zvolí si, kam výstup uložit.

Na obrázku [3] jde vidět, jak aplikace vypadá.



Obrázek 3: Aplikace pro generování vzorků z grafu.

6 WikiSearch

Další aplikace, kterou jsou vytvořil, nese pracovní název WikiSearch. Mým cílem bylo pro data najít reálné využití, a vznikl tak vyhledávací našeptávač. Vyhledávání ovšem nefunguje jako u Google nebo samotné wiki. Nehledá se v textu dokumentů. Místo toho uživatel vyhledává v titulcích stránek. Během vyhledávání se uživateli nabízí titulky, které obsahuji zadaný podřetězec. Pořadí není abecedně, ale podle indexu výskytu, a první jsou tedy výsledky, které výrazem začínají. Vyhledaný výraz (titulek) se přidá do kolekce hledaných. Výsledek jsou pak opět titulky, ale takové, které nejvíce souvisí se zadáním. To jsou takové, které mají v grafu společnou hranu, tedy takové, které na sebe navzájem odkazují. V úvahu je brána také váha hrany a výsledky jsou podle váhy sestupně seřazené. Mezi prvními jsou takové stránky, které byly odkazovány vícekrát. Pokud byla stránka výsledku odkazována z více vstupních parametrů, součet jejich vah se navíc vynásobí jejich počtem, což ještě zvýší hodnocení stránky. Výsledek má také tlačítko, které jej přidá mezi vyhledávané. Uživatel tak může zpřesňovat svoje zadání a získat představu o souvisejících tématech. Každý výsledek navíc uvádí odkaz na originální článek na Wikipedii.

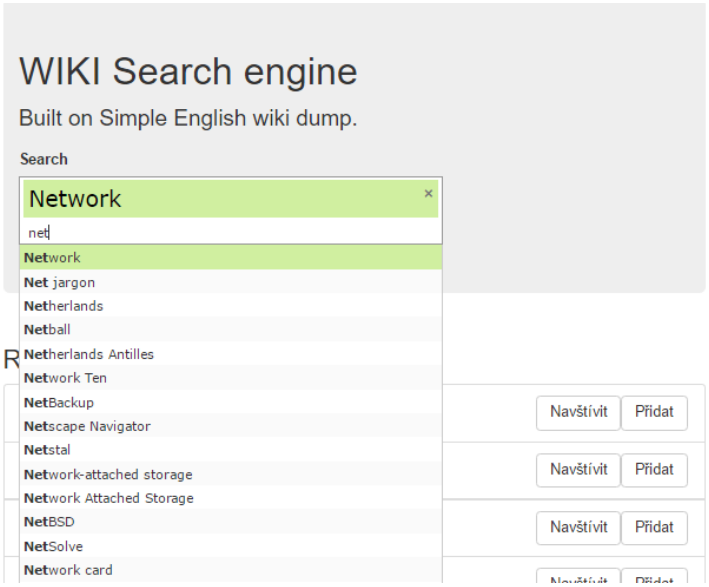
Pojďme se podívat na příklad. Zadáme do hledání Pet a výsledky jsou Domestication (6) a dále výpis zvířat Dog (6), Rat(6), Fish(4) a Cat(4). Přidáme-li do vyhledávaných pojmů první nabízený pojem Domestication, získáme výsledky Dog (32) , Cat (20) a Guinea pig (16). Výsledek se blíží očekávání, dostali jsme seznam nejčastěji domestikovaných mazlíčků.

Podobně pro výrazy Europe a Euro dostaneme přibližně seznam evropských států, přesněji politických uskupení, kde se platí eurem - Evropská unie, Francie, Finsko, Řecko.

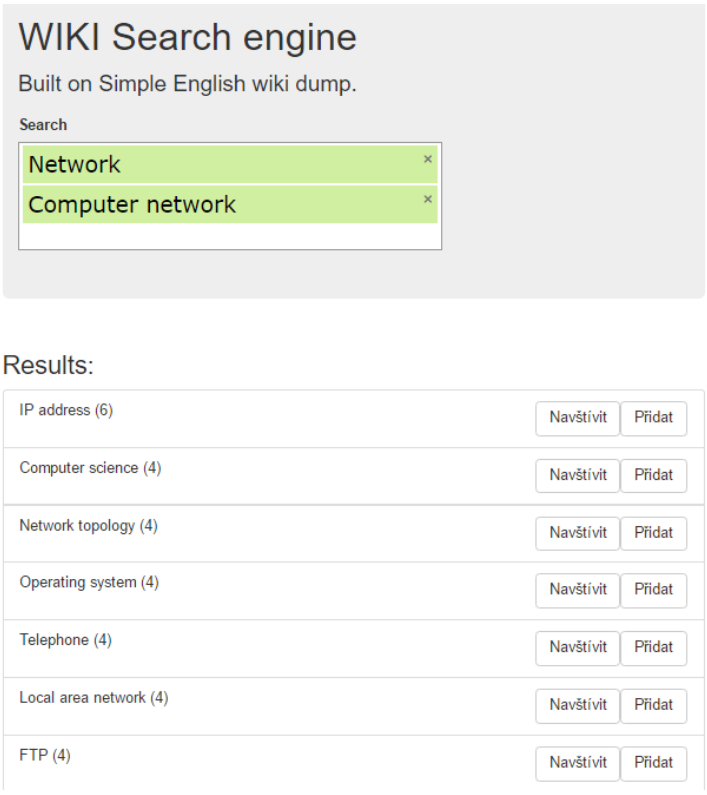
Pro výrazy Network a Computer Network jsou výslednými pojmy IP address, Computer Science, Network topology...

Ve výsledcích lze najít náznak původního záměru a pokud se budeme snažit, získáme výsledek, jaký bychom očekávali. Nedá se však říct, že by tato pomůcka v této formě byla k něčemu opravdu užitečná.

Následují obrázky [4] a [5], kde lze vidět, jak vypadá uživatelské rozhraní aplikace. Aplikace je postavena v technologii ASP.NET MVC a vzhled vychází z CSS frameworku Bootstrap.



Obrázek 4: Našeptávání výsledků



Obrázek 5: Výsledky vyhledávání.

7 Analýza sítě

Abych mohl provést další zpracování a analýzy grafu, rozhodl jsem se změnit prostředí a místo dosavadního .NET Frameworku jsem využil jazyk Python.

Než jsem Python vybral, udělal jsem si na internetu průzkum, který jazyk je k tomuto účelu nejvhodnější. Našel jsem různá doporučení, například často zmiňovaný R-lang. Python byl doporučován nejčastěji kvůli komplexní knihovně NetworkX a možnosti spouštět céčkové knihovny Graph-tools a iGraph. Na internetu je také dostupné velké množství návodů a diskuzí týkajících se kombinace jazyka Python a mého tématu.

7.1 Načtení dat z CSR

První krok pro práci s daty je načtení ze souboru. Otevírat chceme sparse matici, o které jsem psal v předchozích kapitolách. Jedná se o CSV soubor, ve kterém je uložena tabulka ve formátu CSV. Velmi často používanou knihovnou pro Python je SciPy [5]. Název je odvozen od slova Science (věda) a jde o soubor open-source nástrojů pro matematické, věděcké a inženýrské úkoly v Pythonu. V tomto programu využijeme především jeho schopnost pracovat s CSR maticemi. Mimo jiné podporuje také další podobné formáty a převod mezi nimi, nebo převod do plné matice (dense matrix). Napsal jsem tedy 2 základní metody. Jedna slouží pro načtení do csr_matrix, druhá k vrcholům načte jejich titulky, aby se výsledky lépe zkoumaly. SciPy ale podporuje více formátů, než jen CSR a rozhodl jsem se prozkoumat některé alternativy.

7.2 CSR a jeho alternativy

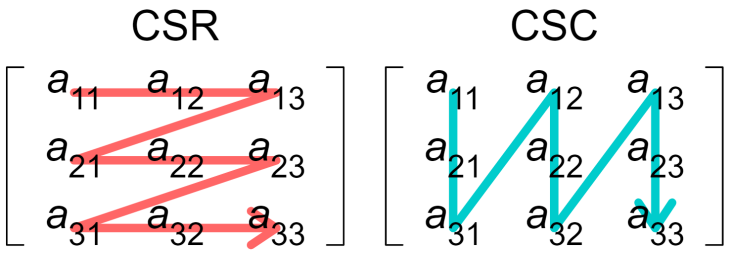
Přímo knihovna SciPy umí pracovat s několika formáty sparse (řídkých) tabulek. Jejich kompletní seznam je: CSR, CSC, COO, BSR, DIA, DOK, LIL. Implementuje také metody, které dokáží z libovolného formátu převézt tabulku na jakýkoli jiný formát. Příklad může být převod z CSR na CSC.

```
csc = csr.tocsc()
```

Převod je navíc velmi rychlý a programátor tak může využít takový formát, který zrovna v daný moment potřebuje s ohledem na optimalizaci. Co ale jednotlivé zkratky znamenají a jaký formát se za nimi skrývá? A jaký vybrat formát podle výhod které nabízí? Prostudoval jsem jednotlivé formáty, abych mohl vyhodnotit, zda byla CSR správná volba. Při tomto výzkumu jsem se převážně opíral o dokumentaci knihovny SciPy [5].

7.2.1 CSR

Zkratka znamená Compressed Sparse Row. Tabulka je tvořena třemi seznamy. První seznam obsahuje všechny nenulové hodnoty z tabulky. Druhý seznam obsahuje řádkový index pro každou



Obrázek 6: Výsledky vyhledávání.

hodnotu z první tabulky a je tak stejně dlouhý jako je seznam hodnot. Třetí seznam pak udává indexy do prvních dvou. Na těchto indexech začíná nový řádek tabulky. Délka seznamu tedy odpovídá dimenzi tabulky.

Výhody tohoto formátu popisuje přímo dokumentace. Je to především efektivní získání celých řádků (row slicing) a efektivní operace pro sčítání a násobení matic. Nevýhodou jsou naopak výpočetně drahé úpravy, které mění strukturu tabulky, a drahá operace pro přístup k sloupečkům. Tuto výhodu řeší CSC.

7.2.2 CSC

Compressed Sparse Column. Jde o obdobu CSR, ovšem zatímco CSR postupuje po řádcích, CSC postupuje po sloupcích. Rozdíl je patrný na obrázku [6].

Výhody a nevýhody jsou stejné jako CSR, s tím rozdílem, že zde je drahá operace pro získání řádku a levné naopak získání sloupečku. Protože jsem této vlastnosti využil později v algoritmu LPA, provedl jsem malý výkonnostní test. Zajímalo mě, jak velký je tento rozdíl, a změřil jsem, jak dlouho trvá 1000 přístupů ke sloupečkům metodou getCol. Test jsem provedl na tabulce CSC a CSR a rozdíl byl výrazný. Zatímco z tabulky CSC těchto tisíc operací trvalo pouze dvě vteřiny, dokonce včetně převedení CSR na CSC, tak stejný úkol v tabulce CSR trval 12 vteřin. Pokud bych tento test dělal na celé tabulce se kterou pracuji, byl by poměr asi 37 minut k 21 vteřinám. Je tedy velmi důležité zvolit správný formát pro daný úkol.

7.2.3 COO

Coordinate list je to, co většina programátoru napadne jako první nejjednodušší řešení. Jedná se o seznam, kde každou hodnotu v tabulce reprezentuje tuple, který obsahuje index řádku, index sloupce a hodnotu. Tento formát je vhodný především pro budování tabulky. K tomu je uzpůsoben mimo jiné tím, že umožňuje duplicitní záznamy. Při následném převodu do jiného formátu se duplicity sečtou, což může být užitečné. A právě převod do formátu CSR a CSC je v dokumentaci popisován jako velice rychlý. Negativní vlastností COO je pak špatná podpora slicingu (přístup k řádkům nebo sloupcům) a aritmetické operace. Všechny tyto vlastnosti tedy předurčují k postupu, kdy COO využijeme jako konstrukční formát a ten následně převedeme do CSR, nebo CSC.

V mojí práci jsem tento formát využil při konstukci sítě v knihovně NetworkX. Parametry této metody totiž přesně odpovídají tomu, co COO obsahuje. Tedy tři pole s hodnotami, indexy řádků a indexy sloupečků.

7.2.4 BSR

Block Compressed Row je formát, který přímo vychází z formátu CSR. Ovšem namísto číselných hodnot je složený z menších matic. Tyto matice mají přesně daný rozměr a ten musí být bezezbytkovým dělitelem rozměrů celé matice. Má tak velmi podobné vlastnosti jako CSR, ale hodí se na některé speciální případy, jako například počítání s vektory. V těchto případech pak dokáže poskytnout vyšší výkon.

7.2.5 DIA

Formát nazývaný Diagonal patří mezi formáty, které jsou užitečně v určitých speciálních situacích. V tomto případě je zaměření na diagonální matice, což jsou takové, které mají většinu nebo všechny hodnoty na diagonále. V takovém případě pak lze matici o velikosti n*n uložit jako jediné pole o délce n.

7.2.6 DOK

Dictionary of keys, tak se nazývá formát, který hodnoty ukládá jako mapu. Klíč této mapy jsou koordináty a hodnota je hodnota uložená na daném místě v tabulce. Nuly se v mapě nenachází. Jedná se tak o další formát, který je vhodný především pro budování nebo přístup k jednotlivým hodnotám. Dokumentace dále zdůrazňuje efektivní převod na COO formát.

7.2.7 LIL

List of lists je poslední formát, který knihovna nabízí. A i tento se řadí jako formát, který je spíše vhodný ke konstrukci tabulky. Hodnoty jsou uložený zvlášť pro každý řádek tabulky. Každý řádek pak obsahuje seznam s hodnotami a jejich umístěním na řádku. Jeho výhody jsou pouze v snadném přístupu k řádkům a nenákladné změně ve struktuře tabulky. Dokumentace také upozorňuje, že na velké tabulky se více hodí COO.

7.2.8 Podobná využití

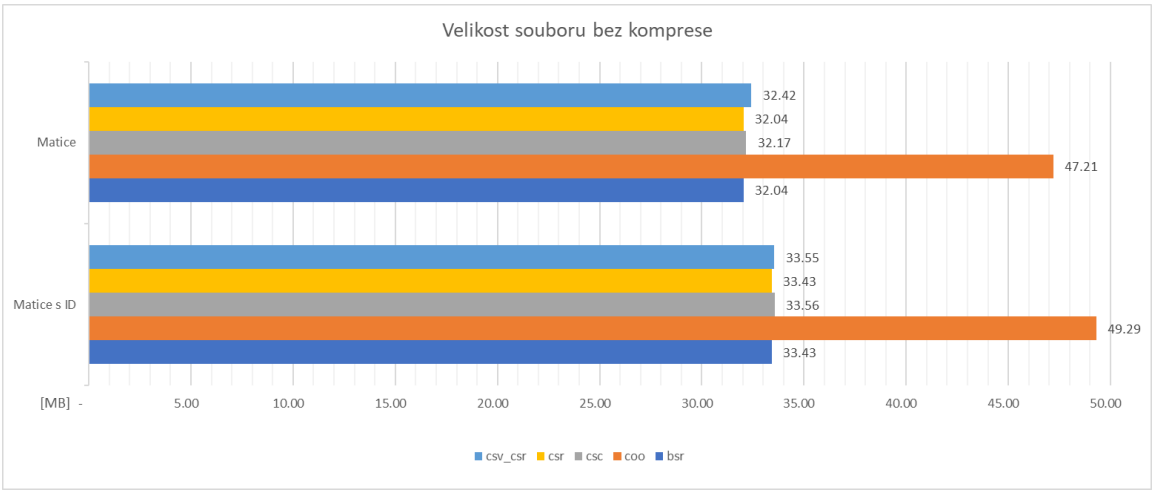
Podobný formát jako je tento jsem použil i já, když jsem implementoval uložení tabulky v paměti programu v C#. V objektové reprezentaci jsme si uložil každou hodnotu do dictionary podle jejího indexu na řádku. Celou tuto dictionary jsem uložil do druhé dictionary podle indexu řádku. Tento formát byl pro mě výhodný právě proto, že jsem tabulku teprve budoval. Navíc jsem jí tvořil po řádcích, jelikož na každém řádku byla jedna stránka a ve sloupcích všechny odkazy na stránky. Vznikala tedy po řádcích. Také přístup na konkrétní pozici je rychlý, protože

jde o prohledání dvou indexovaných seznamů. Nejde ovšem o přímé srovnání, protože zde je řeč o standardním formátu pro uložení, zatímco já jsme tento tvar použil pouze pro organizaci objektů v paměti za pomoci dostupných kolekcí. Výhodou tohoto řešení je, že v případě potřeby se dá struktura velice snadno přeorganizovat. Například přeskupit tak, aby byl nejprve přístup podle sloupečku, až poté podle řádku. Nebo pro získání tvaru COO stačilo načíst všechny hodnoty do seznamu, protože objekty obsahují informaci o pozici v tabulce. Také DOK bych vytvořil pouze přenesením ze dvou do jedné dictionary. Některé tyto transformace jsem využil pro pohodlnější práci, nebo zvýšení výkonu. Vyplývá z toho, že programátor by si měl uvědomit, který formát je pro něj ideální v daném případě využití.

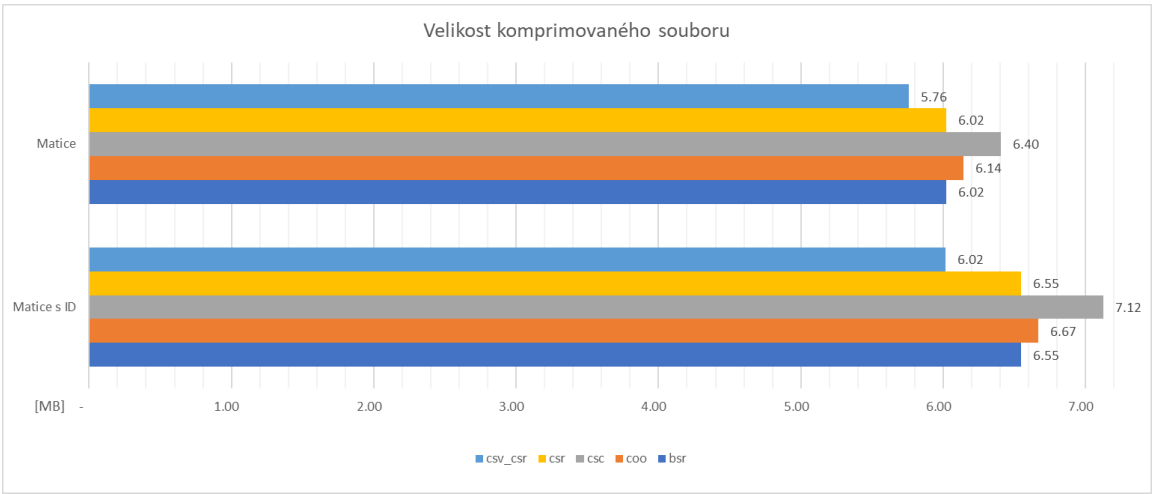
7.2.9 Porovnání velikosti

Po tomto průzkumu jsem si potvrdil, že volba CSR formátu byla pro má data vhodná. Zajímalo mě však, jestli je tento formát efektivní také v otázce velikosti výstupu. Využil jsem toho, že díky knihovny SciPy mohu tabulku převádět mezi formáty, a toho, že umožňuje některé formáty uložit do souboru. Vyřadit jsem musel formát DIA, který se nepodařilo převést kvůli nedostatku paměti - během převodu se tabulka převádí na plný tvar. Dále jsem vynechal DOK a LIL, protože pro tyto formáty není implementováno uložení do souboru. K porovnání jsem mohl přistoupit u formátů CSR, CSC, COO, BSR a původnímu CSR formátu, který jsem uložil vlastní metodou. Každý z těchto jsem navíc otestoval ve 4 variantách. Dvě varianty jsou pro komprimovaný soubor a dvě pro nekomprimovaný. Z těchto je pak vždy jedna, kde je uložena pouze matice, a druhá, kde jsem na první řádek vložil seznam ID vrcholů. Bez těchto ID totiž není informace o grafu kompletní, a data tak více odpovídají mým potřebám. Zatímco má implementace je čistě v textové podobě ve formátu CSV, knihovna SciPy ukládá data do formátu NPZ. NPZ je ve skutečnosti ZIP, který můžeme otevřít a podívat se na jeho obsah. Uvnitř najdeme binární soubory typu NPY, ve kterých jsou binárně uloženy proměnné nutné pro sestavení tabulky. Pro každou proměnnou jeden soubor. Z pravidla jsou to shape.npy, format.npy, data.npy a podle formátu indices.npy a indptr.npy, nebo col.npy a row.npy. Mě ovšem zajímala pouze velikost celého souboru. Tuto velikost jsem zanesl do grafu [7], respektive [8] pro komprimovanou verzi.

Při pohledu na výsledky bez komprese [7] jsou si výsledky formátu velice blízké. Pouze COO výrazně převyšuje svojí velikostí a oproti ostatním je skoro o 50 % větší. COO soubor je velký přibližně 50 megabytů, ostatní formáty něco málo přes 33 megabytů. Jen těsně by vyhrál formát CSR, respektive shodně uložený BSR. Ovšem ani CSC ani moje vlastní metoda výrazně nezaostávají. Méně vyrovnaný souboj se odehrává v komprimované verzi [8], kde je úspora mezi nejmenším (6,02 MB) a největším (7,12 MB) formátem 15 % velikosti souborů. Ten největší je CSC. Druhý největší je COO (6,67MB), kterému komprese hodně pomohla, protože správně rozpoznala často se opakující koordináty. Třetí je formát CSR a BSR (6,55MB). Vítězným formátem je tedy textový soubor s mou vlastní strukturou. Tento úspěch bych přisoudil slovníkové kompresi, která má lepší výsledek díky tomu, že celý graf komprimuje jako jeden soubor.



Obrázek 7: Velikost formátů Sparse matice.



Obrázek 8: Velikost formátů Sparse matice s komprimaci.

Z tohoto pokusu jsem vyvodil závěr, že nadále nejvhodnější metodou uložení je vlastní textový formát. Z hlediska velikosti je stejný jako ostatní, v komprimované formě dokonce nejmenší. Za značnou výhodu pak považuji fakt, že soubor je snadno čitelný a použitelný v libovolném prostředí.

7.3 Vytvoření grafu v NetworkX

Teď když máme data načtená v paměti, zrekonstruujeme z nich náš graf. Pro tento úkol jsem zvolil knihovnu NetworkX [6], která je velmi bohatá na funkce, které nabízí v rámci analýzy sítí. K sestavení grafu nám stačí v podstatě 3 řádky kódu. Jeden pro vytvoření grafu, druhý vytvoří správný počet vrcholů a třetí přidá hrany z dat, které jsme si načetli. Chceme-li přidat titulky a původní ID stránky, přidáme je ke grafu jako kolekci atributů. Celou metodu pro vytvoření grafu lze vidět v ukázce [1]. Tato metoda má na vstupu CSR tabulku načtenou ze souboru a k ní také kolekci s ID a Titulky vrcholů. Načtení celého grafu, včetně čtení souboru trvá obvykle asi 27 vteřin. Pokud chceme pracovat s největší komponentou, trvá její získání dalších deset vteřin.

```
def load_networkx_graph(csr_matrix, graph_nodes):
    # create new graph
    G = nx.DiGraph()

    # set nodes
    G.add_nodes_from(range(csr_matrix.shape[1]))

    # add original page ID and Title as attribute. Optional step.
    nx.set_node_attributes(G, graph_nodes[0], 'nodeid')
    nx.set_node_attributes(G, graph_nodes[1], 'nodetitle')

    # conver CSR to COO and create edges of graph
    coo = csr_matrix.tocoo()
    G.add_weighted_edges_from(zip(coo.row, coo.col, coo.data))
    return G
```

Výpis 1: Vytvoření grafu NetworkX z CSR tabulky.

7.4 Analýza v NetworkX

Jednotlivé kroky analýzy jsem sepsal do podkategorií tak, jak jsem je postupně aplikoval. Celý postup odpovídá pythonovému skriptu, který je součástí digitální přílohy. Vybrané funkce jsou obsaženy přímo v textu.

7.4.1 Hustota

Funkcí density získáme údaj o hustotě sítě. Nabývá hodnot mezi 0 a 1, kde 0 znamená graf bez hran, 1 znamená graf úplný. Výsledek metody je v našem případě číslo 0.000125, tedy jde o velmi řídký graf. Tento údaj si můžeme snadno ověřit. Spočítáme si počet hran pro případ úplného grafu. V případě orientovaného grafu nesmíme zapomenout, že úplný graf má hran dvakrát tolik.

$$e_{full} = \frac{n * (n-1)}{2} * 2$$

A hustotu spočítáme jako poměr mezi počtem hran v testovaném grafu a grafu úplném.

$$d = \frac{e}{e_{full}}$$

$$d = 0.000124578$$

7.4.2 Hledání nejkratší cesty

V případě hledání nejkratší cesty už bychom si s matematikou nevystačili. Tento problém už budeme muset řešit algoritmizací. Algoritmus ale nemusíme psát sami, využijeme místo toho funkci knihovny networkX. Vyzkoušíme si to na dvou stránkách s ID 0 a 15.

```
nx.shortest_path(G, source=0, target=15)
nx.shortest_path(G, source=15, target=0)
```

Nejkratší cesta z vrcholu 0 do vrcholu 15 je 1 : [0,15]. To znamená, že z vrcholu 0 vede hrana přímo do vrcholu 15. Když vrcholy prohodíme, je ale nejkratší cesta délky 3 : [15, 629, 1, 0]. Abychom došli z vrcholu 15 do vrcholu 0, musíme navštívit a projít přes nejméně 2 další vrcholy 629 a 1. Ověřili jsme si, že graf jsme správně vytvořili jako orientovaný.

7.4.3 Souvislost grafu

Graf je souvislý tehdy, pokud pro každou dvojici vrcholů existuje cesta z jednoho do druhého. Zkoumáme orientovaný graf, proto rozlišujeme slabou a silnou souvislost. V případě silné souvislosti musí pro každou dvojici vrcholů existovat cesta z vrcholu 1 do vrcholu 2, ale také obráceně. V případě slabé souvislosti nás zajímá, zda by existovala cesta mezi vrcholy, pokud bychom graf převedli na neorientovaný. Obě tyto otázky zodpovíme zavoláním funkce.

```
nx.is_strongly_connected(G)
nx.is_weakly_connected(G)
```

V obou případech se dozvíme, že graf souvislý není a graf obsahuje více než jednu komponentu.

7.4.4 Hledání komponent

Když už víme, že zkoumaný graf není souvislý, můžeme využít další funkci pro hledání všech komponent. Přesněji řečeno dvě metody, protože i zde platí, že komponenty mohou být propojené silně nebo slabě.

```
nx.number_strongly_connected_components(G)
nx.number_weakly_connected_components(G)
```

Výsledek první metody pro silně propojené komponenty vrací číslo 77,964. U slabě propojených komponent bychom měli očekávat menší počet, protože kritéria jsou mírnější. Výsledek je 659 komponent. Počet komponent je sice zajímavý, ale užitečnější budou funkce, které nám vrátí samotné komponenty.

```
components = nx.weakly_connected_components(G)
components = nx.strongly_connected_components(G)
```

Každá komponenta je reprezentovaná seznamem vrcholů. Z těch si můžeme například vypsat četnost komponent se stejným počtem vrcholů. Tedy dozvíme se, jak velké jsou nalezené komponenty. Velikosti a počet takových komponent nalezneme v tabulce [2] a [3].

7.4.5 Podgraf

V tabulce slabě spojených komponent lze vidět, že graf obsahuje jednu velkou komponentu, která zahrnuje většinu vrcholů. Poté několik menších komponent a nakonec velký počet malých komponent o velikosti jednoho až dvou vrcholů. V případě silného propojení je situace ještě extrémnější. Pro další práci jsem se rozhodl pokračovat jen s největší silně spojenou komponentou. Tedy z původního grafu vzniká nový graf s počtem vrcholů 102,080 a 3,032,724 hran. Můžeme si také ověřit, že metody pro test souvislosti už obě vrací True.

```
largest_component = max(components, key=len)
largest_subgraph = G.subgraph(largest_component)
```

7.4.6 Write metody

Výsledný graf je vhodné uložit, abychom s ním mohli příště začít pracovat. Knihovna NetworkX nabízí poměrně široké možnosti pro export do souboru. Můžeme si vybrat z těchto formátů: GEXF, GML, Pickle, GraphML, JSON, LEDA, YAML, SparseGraph6, Pajek, GIS Shapefile a několika druhů listů. Bohužel, když jsem se pokusil graf exportovat, zjistil jsem, že většina těchto metod nefunguje. Některé po chvíli zhavarují na nedostatek paměti, jiné trvaly velmi dlouho bez

Tabulka 2: Silně spojené komponenty

Velikost komponenty	Počet komponent
1	77055
2	639
3	128
4	44
5	18
6	21
7	13
8	9
9	6
10	9
11	4
13	1
15	1
18	1
19	2
20	4
23	2
25	3
31	1
36	1
37	1
46	1
102080	1

Tabulka 3: Slabě spojené komponenty

Velikost komponenty	Počet komponent
1	63
2	544
3	29
4	6
5	5
6	4
7	1
9	3
11	1
14	1
21	1
180584	1

výsledku. Jedna se sice jevila být v pořádku, ale výsledný soubor byl prázdný. Jiné metody mají nejspíš vyšší požadavky na data nebo parametry grafu, takže skončily chybou. Pouze jediná

testovaná metoda byla úspěšná a povedlo se jí v čase 122 sekund graf exportovat do souboru a uložit na disk. Jedná se o metodu write_gml. Bohužel už se ale nepodařilo graf znovu načíst metodou load. Načtení grafu se nepodařilo ani ze souboru GEXF, který jsem vytvořil.

7.4.7 Šířka grafu

Zápis do souboru není jediná metoda, která má problém s rozsáhlým grafem jako je tento. Spíše naopak. Platí to také pro šířku grafu.

```
diameter = nx.diameter(largest_subgraph)
```

Šířka grafu je nejdelší ze všech nejkratších cest. Pokud bychom ho chtěli vypočítat naivní metodou, tedy kdybychom spočítali všechny nejkratší cesty a vybrali tu nejkratší, trval by podle odhadu výpočet 3732 dní. Pro tento odhad jsem změřil určitý náhodný vzorek vrcholů, nemusí odpovídat skutečnosti. Navíc je málo pravděpodobné, že by knihovna využila takto jednoduchou neoptimalizovanou metodu. Jde jen o příklad pro představu.

7.4.8 Statistiky vrcholů

Statistika vrcholů nám prozradí mnoho o tom, které vrcholy jsou v grafu významnější než jiné. Podíváme se postupně na 6 tabulek s hodnotami nejvyšších stupňů vrcholů, zvlášť vstupních a výstupních stupňů, hodnotu Eigenvector centrality a nakonec huby a autority algoritmu HITS.

Z těchto hodnot jsou poslední dvě metody výpočetně náročnější. Fungují iterativní metodou, dokud není výsledek ustálený nebo nedojde k limitu počtu iterací. Především v případě HITS jsem s tímto měl problém, protože výpočet trval velmi dlouho. Chci tedy upozornit, že NetworkX u tohoto algoritmu nabízí kromě funkce hits() také variantu hits_scipy(). Tato funkce využívá pro výpočet knihovnu scipy a je tak výrazně rychlejší.

Začneme tabulkou, ve které jsou vypsány nejvyšší stupně vrcholů [4]. Na prvním místě je s velkým náskokem článek nazvaný Wikipedia:Stub. Pokud bychom si tento článek přečetli, dočteme se, že Stub je něco jako velmi krátký článek, často pouze hrubý popis vytvořený ze šablony. Takový článek je vždy označený jako Stub, což by se v češtině dalo přeložit jako pahýl. Zároveň je ale v každém takto označeném článku odkaz na zmíněný článek, který Stub vysvětluje. Proto se jedná o nejčastěji odkazovanou stránku Wikipedie. Podobně je na tom stránka Help:Template, tedy článek popisující samotné fungování wiki. Ale na dalších pozicích už nalezneme převážně geografické celky. Těm dominuje Amerika, Francie nebo Anglie.

Přejdeme-li na další tabulku, ve které jsou vrcholy s největším vstupním vrcholem [5], zjistíme, že tabulky jsou si velmi podobné. A když si porovnáme počty v těchto dvou tabulkách, zjistíme, že jsou si taky velice blízké. Vidíme tedy seznam vrcholů, na které se nejčastěji odkazuje. První příčku už jsme si vysvětlili. Ale i další pozice jsou zde z pochopitelných důvodů. Pokud je Wikipedie plná stránek třeba o řekách, bude mnoho z nich v Americe, protože je to

jeden z největších územních celků. A dá se předpokládat, že každá taková řeka bude obsahovat odkaz s informací, že leží v Americe. Podobný princip můžeme očekávat u měst, významných osobností a dalších témat.

Pokud odečteme tabulku vstupních vrcholů od tabulky vrcholů, můžeme tušit, že tabulka vstupních vrcholů [5] bude vypadat naprosto odlišně. Stránky, které zde najdeme, jsou také zcela jiného charakteru. Jedná se o různé indexy, tedy stránky, které obsahují nějaký seznam. Například první stránka List of years obsahuje seznam s roky 800 až do budoucího 2100. Dále najdeme seznam anglických slovních spojení, seznam významných úmrtí. Konec seznamu by některé mohl překvapit - jde o regiony Francie. A každý takový region má na stránce seznam ostatních regionů.

Právě tato hustá síť francouzských regionů se projeví v tabulce [7] seřazené podle hodnoty centrality vrcholu, což je hodnota, ze které se počítá PageRank. Vytvořila zde jev známý pod označením rank sink. Stalo se to, že těchto několik stránek má navzájem tolik propojení, že jejich PageRank je výrazně vyšší než PR všech ostatních.

Podobně nešťastně dopadla i další statistická metoda zvaná Hyperlink-Induced Topic Search (HITS). Ta se snaží najít v grafu autority a huby. To jsou významné vrcholy v grafu podle toho, jak byly v minulosti chápány internetové stránky. Huby jsou rozcestníky, které vedou na různé stránky. Autority jsou stránky, které se často objevují v hubech. V našem případě jsou obě tabulky monotematické a obsahují stránky z jedné jediné komunity. Jsou to stránky, které popisují všechny jazykové mutace Wikipedie.

7.4.9 Hledání komunit

Při hledání komunit jsem také nezaznamenal žádný úspěch. Přestože knihovna nabízí několik různých algoritmů pro hledání komunit, ani jediná z nich nedosáhla alespoň částečného výsledku. Vyzkoušel jsem tyto metody: Denrogram, Fast Greedy, Kernighan–Lin bipartition, Girvan-Newman, Async label propagation. Při snaze dosáhnout úspěchu jsem graf převedl na neorientovaný a neválený graf. Vytipované algoritmy jsem spustil a dal jim dostatek času. Vybral jsem Fast Greedy a Label Propagation. Každý z těchto algoritmů běžel na mém počítači přibližné 5-7 dní, ovšem bez výsledku.

Po tomto neúspěchu jsem usoudil, že pokud nedělám nějakou chybu já, bude pravděpodobně problém v knihovně. Autor či kolektiv autorů nejspíš nepředpokládal výpočet nad takto velkými grafy. Rozhodl jsem se zkusit jinou knihovnu. O tom budu psát v následující kapitole.

7.5 Hledání komunit pomocí iGraph

Protože při hledání komponent za pomoci knihovny NetworkX jsem nebyl úspěšný, hledal jsem jiný způsob, potažmo jinou knihovnu.

Předchozí knihovna NetworkX je napsaná v Pythonu. Tento jazyk je interpretovaný, a z toho důvodu nepatří mezi ty nejvýkonnější. Naproti tomu knihovna iGraph je napsaná v jazyce C.

Tabulka 4: Node degree

Degree	Title
57248	Wikipedia:Stub
15548	United States
10076	France
7377	Americans
7300	Help:Template
5685	Departments of France
5097	Time zone
4976	Communes of France
4744	England
4603	United Kingdom
4599	International Standard Book Number
4525	Japan
4500	Regions of France
4492	Daylight saving time
4458	Germany
4393	City
3427	Italy
3380	Canada
3378	Australia
3256	Association football

Tabulka 5: Node in degree

Degree	Title
57232	Wikipedia:Stub
15126	United States
9734	France
7328	Americans
7299	Help:Template
5529	Departments of France
5090	Time zone
4956	Communes of France
4572	England
4565	International Standard Book Number
4467	Daylight saving time
4446	Regions of France
4329	United Kingdom
4289	City
4270	Japan
4167	Germany
3202	Italy
3202	Association football
3140	Canada
3081	Australia

Kompiluje se tak do strojového kódu a měla by díky tomu lépe využít výkon počítače. Tento program se pak volá z prostředí Pythonu. O porovnání těchto dvou knihoven jsem našel článek [7], kde je iGraph prezentován jako až desetkrát rychlejší.

Kvůli tomu je také trochu složitější instalace. V případě unixového prostředí to nejspíš není velký rozdíl, protože kompilace probíhá automaticky při instalaci v PIPu. Ale na Windows je proces instalace komplikovanější. Programátor si totiž musí stáhnout zkompilovaný program z internetu a do Pythonu nainstalovat.

Sestavit graf je pak velmi podobné v obou knihovnách. A tak se rychle dostaneme k hledání komunit. Pro porovnání jsem zvolil metodu Label propagation, jelikož se od ní dá předpokládat nejkratší čas k dosažení výsledku. Když jsem si funkčnost skriptu včetně uložení výsledku vyzkoušel na příkladu, spustil jsem výpočet.

I zde jsem dal počítači dostatek času na výpočet a počítač jsem nechal pracovat přesně sedm dní. Ani tady jsem se ale výsledku nedočkal a výpočet jsem musel ukončit a hledat jiný způsob.

7.5.1 Hledání komunit Graph-Tools

Poté, co jsem nebyl úspěšný ani v NetworkX, ani iGraph, nabízela se ještě třetí možnost. Mohl jsem vyzkoušet framework Graph-tools, který jeho autoři popisují [7] jako ještě výkonnější. A to

Tabulka 6: Node OUT degree

Degree	Title
1388	List of years
1190	Wikipedia:Basic English combined wordlist
1182	Deaths in 2013
969	List of people from Texas
929	Wikipedia:List of articles all languages should have
927	2012 in movies
835	List of cities in Iowa
804	Deaths in 2012
797	2006
787	2008
775	Wikipedia:VOA Special English Word Book
750	List of people from New Jersey
749	2007
730	Deaths in 2011
702	April
695	Bouret-sur-Canche
695	Buissy
695	Campigneulles-les-Grandes
695	Dourges
695	Foncquevillers

Tabulka 7: Eigenvector centrality

Degree	Title
0.05082672	France
0.05029585	Departments of France
0.05001664	Wikipedia:Stub
0.04981479	Communes of France
0.04978930	Regions of France
0.04924301	Caen
0.04879392	Lisieux
0.04879032	Bayeux
0.04879002	Isigny-sur-Mer
0.04878943	Agy
0.04878943	Arganchy
0.04878943	Arromanches-les-Bains
0.04878943	Asnelles
0.04878943	Barbeville
0.04878943	Bazenville
0.04878943	Bernesq
0.04878943	Blay
0.04878943	Bricqueville
0.04878943	Cahagnolles
0.04878943	Campigny, Calvados

především v případě použití OpenMP varianty, která dokáže využít všechna procesorová jádra. Rozhodl jsem se ale, že se touto cestou nevydám. Místo toho jsem se rozhodl implementovat vlastní metodu, nad kterou budu mít větší kontrolu.

7.6 Vlastní metoda pro hledání komunit

Testované knihovny mají jednu společnou a nepříjemnou vlastnost. Programátor nemá žádnou šanci dozvědět se, jak daleko algoritmus pokročil. Mohlo se tak stát, že jsem program zastavil pár minut před tím, než by došel k výsledku.

Abych mohl udělat nějaký závěr, potřeboval bych odhadnout, jak dlouho by trvalo komunity nalézt. Toho jsem docílil tím, že jsem napsal vlastní implementaci algoritmu. Tím mám kontrolu nad jeho během, mohu tedy zkoumat průběh řešení. Zvolil jsem opět stejný algoritmus, tedy Label Propagation. Ten má také tu výhodu, že běží ve smyčce, která v každé iteraci zlepší výsledek. Mohu se rozhodnout ho v některé fázi zastavit a získal bych alespoň nějaký výsledek. Nevýhoda tkví v tom, že v zahrnuje určitou dávku náhodnosti a pro stejný graf neprodukuje

Tabulka 8: HITS: Autority

Value	Title
0.0103027	Wikipedia
0.0102997	Minangkabau Wikipedia
0.0102716	List of Wikipedias
0.0102240	Serbo-Croatian Wikipedia
0.0102228	Catalan Wikipedia
0.0102228	Tagalog Wikipedia
0.0102226	Chinese Wikipedia
0.0102225	Ukrainian Wikipedia
0.0102224	Italian Wikipedia
0.0102224	German Wikipedia
0.0102223	French Wikipedia
0.0102222	Spanish Wikipedia
0.0102222	Portuguese Wikipedia
0.0102222	Polish Wikipedia
0.0102222	Esperanto Wikipedia
0.0102222	Vietnamese Wikipedia
0.0102222	Indonesian Wikipedia
0.0102222	Waray Wikipedia
0.0102222	Japanese Wikipedia
0.0102221	English Wikipedia

Tabulka 9: HITS: Huby

Value	Title
0.5574332	List of Wikipedias
0.0085551	Belarusian Wikipedia
0.0085095	Punjabi Wikipedia
0.0047133	Wikimedia Foundation
0.0045247	English Wikipedia
0.0043976	Wikipedia
0.0043905	Simple English Wikipedia
0.0043450	Chinese Wikipedia
0.0043450	Indonesian Wikipedia
0.0043448	German Wikipedia
0.0043448	Bosnian Wikipedia
0.0043448	Croatian Wikipedia
0.0043448	Serbian Wikipedia
0.0043448	Swedish Wikipedia
0.0043448	Arabic Wikipedia
0.0043444	Alemannic Wikipedia
0.0043000	Spanish Wikipedia
0.0043000	Japanese Wikipedia
0.0043000	Esperanto Wikipedia
0.0043000	Tagalog Wikipedia

unikátní řešení. Nedá se tedy snadno otestovat. Správnou funkci jsem ověřovat tak, že jsem na malých grafech provedl ruční kontrolu s ohledem na náhodně vybrané pořadí, se kterým počítal počítač.

7.6.1 Label Propagation - popis algoritmu

Při implementaci jsem se nesnažil napodobit implementaci knihoven. Nevytvářel jsem tedy žádnou objektovou interpretaci grafu. Místo toho jsem pracoval s maticí sousednosti, kterou už máme načtenou. Ponechal jsem jí ve formě csr_matrix z knihovny SciPy, která poskytuje vysoký výkon. Protože v programu v každé iteraci přistupuji vždy k celému řádku a celému sloupci, využil jsem výhodu formátu CSR a zároveň výhody CSC. CSR poskytuje rychlý přístup k řádkům a CSC rychlý přístup ke sloupcům. Převod za mě zajistila metoda SciPy `csr.tocsc()`.

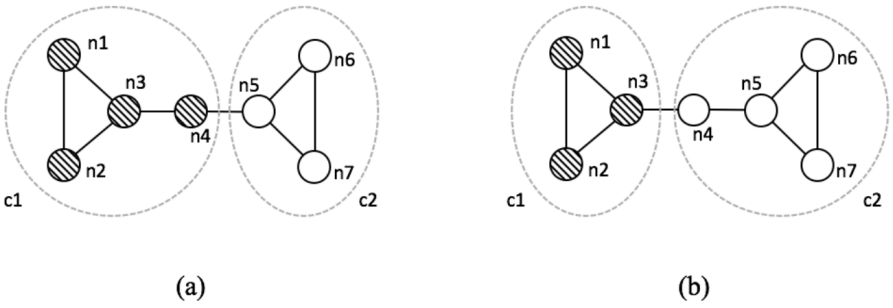
Algoritmus Label propagation pak probíhá následovně. Prvním krokem je přiřazení každému vrcholu počáteční label. Zvolil jsem ID daného vrcholu a uložil ho do pole. V dalším kroku [2] zvolíme náhodné pořadí pro zpracování všech vrcholů. V tomto pořadí nyní pro každý vrchol zjistíme, jaký label se nejčastěji objevuje u jeho sousedů. Nejčastější label se stane novým labelem tohoto vrcholu. *Pokud existuje více možností se stejnou hodnotou, vybírá se náhodně. Během tohoto si zaznamenávám počet změn v kolekci labelů. Pokud nebyla provedena žádná změna, nastal konec algoritmu. Jinak proces proces opakujeme od kroku 2.

Celou implementaci implementaci lze vidět zde: [2]

7.6.2 Hledání výsledku LPA

Výsledky jsem si v každém kroku uložil, abych mohl prozkoumat průběh výpočtu. Sledoval jsem také počet změn a trvání každé iterace. Jedna iterace trvala průměrně 50 sekund. Počet změn se postupně snižoval a poměrně velké překvapení pro mě bylo, když po čtyřiceti minutách program úspěšně dosáhl svého konce. Bohužel jsem se radoval předčasně a rychle jsem si uvědomil, že jsem v algoritmu udělal chybu. Vynechal jsme v něm jeden podstatný krok. Tento krok jsem v popisu LPA výše označil hvězdičkou. Když jsem tuto chybu opravil, konečně jsem vytvořil stejné podmínky jako v případě knihoven networkX a iGraph. Tedy vytvořil jsem algoritmus, který s mými vstupními daty běží v podstatě donekonečna. Tentokrát jsem ale v roli, kdy mohu průběh pečlivě prozkoumat a pochopit důvod.

Jde právě o prve zapomenutý krok výpočtu. Když vrchol vybírá nejčastěji se vyskytující label svých sousedů a nastane situace, že více labelů má stejnou nejvyšší četnost, vybere jeden z těchto labelů náhodně. Dochází k jevu, který se dá dobře představit na následujícím obrázku [9], který jsem si vypůjčil ze článku o odvozeném algoritmu roLPA [[4]]. Na obrázku vidíme příklad grafu ve dvou situacích. Vrchol n4 je v situaci (a) součástí komunity c1 a v situaci (b) je součástí komunity c2. Ve chvíli, kdy se v další iteraci pokusíme pro tento vrchol vybrat vhodný label, budeme mít opět na výběr c1 i c2. Pokud náhoda zvolí jiný label než aktuální, znamená to další iteraci. Takto může vrchol balancovat s pohledu statistiky jen omezenou dobu. Ovšem pokud budeme mít těchto situací více, šance na stabilizaci se snižuje.



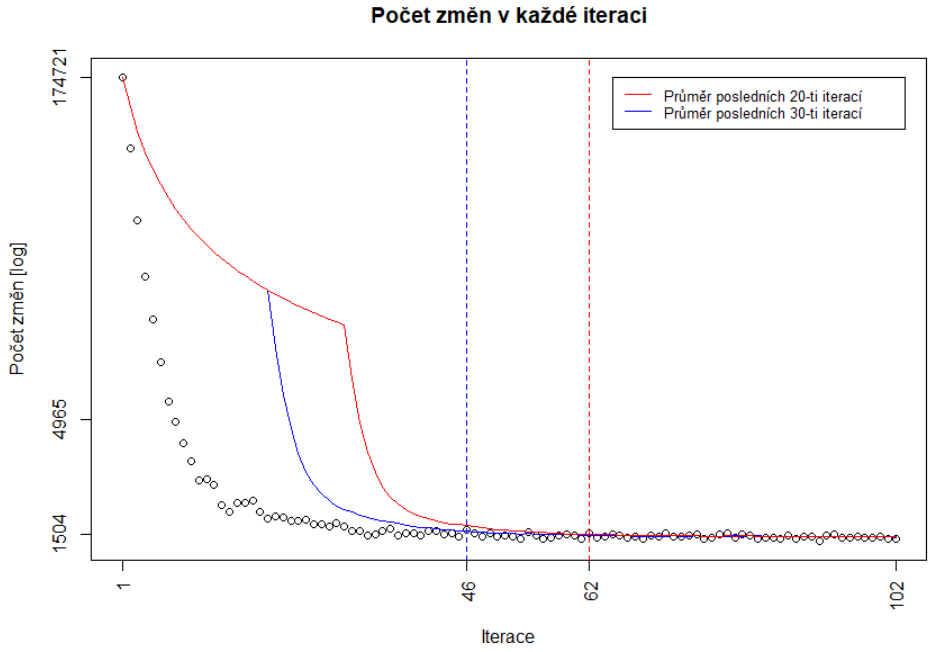
Obrázek 9: Sample network: balancing propagation.

To je právě ta situace, která nastala v našem případě. Lze to vidět na průběhu jednotlivých iterací na obrázku [10]. První iterace provedla změny téměř ve všech vrcholech, druhá v méně než polovině, třetí asi čtvrtině a dále tato hodnota konvergovala zdánlivě k nule. V určitý moment ovšem počet změn přestal klesat a kulminuje kolem hodnoty 1470. Zbylo nám tedy v grafu množství vrcholů, u kterých není jednoznačné do které komunity patří a při každé iteraci balancují mezi více komunitami.

Takových vrcholů je nepochybně více než 1470 a porovnáním iterací 46 až 96 jsem počet spočítal na 2904 vrcholů. To je přibližně dvojnásobek průměrnému počtu změn a odpovídalo

by to binomickému rozhodování, tedy že se tyto balancující vrcholy rozhodují mezi dvěma komunitami. I to jsem početně ověřil a potvrdil tak svůj odhad. Vrcholů, které balancují mezi dvěma komunitami, je 2794. Mezi třemi komunitami balancuje pouze 104 vrcholů a dále pouze 6 vrcholů vybírá ze 4 komunit.

Už tedy víme, proč se PLA jeví jako nekonečný algoritmus. Mohli bychom si položit otázku, proč implementace knihoven NetworkX a iGraph nenabízí nějaký způsob, jak sledovat průběžné výsledky, nebo nenabízí možnost výpočet po čase ukončit. V případě vlastní implementace naštěstí máme vše pod kontrolou a můžeme se tedy sami rozhodnout, kdy výpočet ukončit. Moje řešení tohoto problému spočívá ve sledování průměrného počtu změn za posledních x iterací. Tento průměr jsem zanesl do grafu ve dvou variantách x=20 a x=30. Tímto parametrem mohu regulovat, jak moc stabilizovaný výsledek požaduji. Následně jsem stanovil hranici tehdy, když počet změn byl vyšší než tento průměr. Tímto jsem rozhodl, že za konečný výsledek hledání komunit mohu považovat čtyřicátou šestou iteraci, respektive iteraci 62 pro x rovno 30. Nebo kteroukoli další.



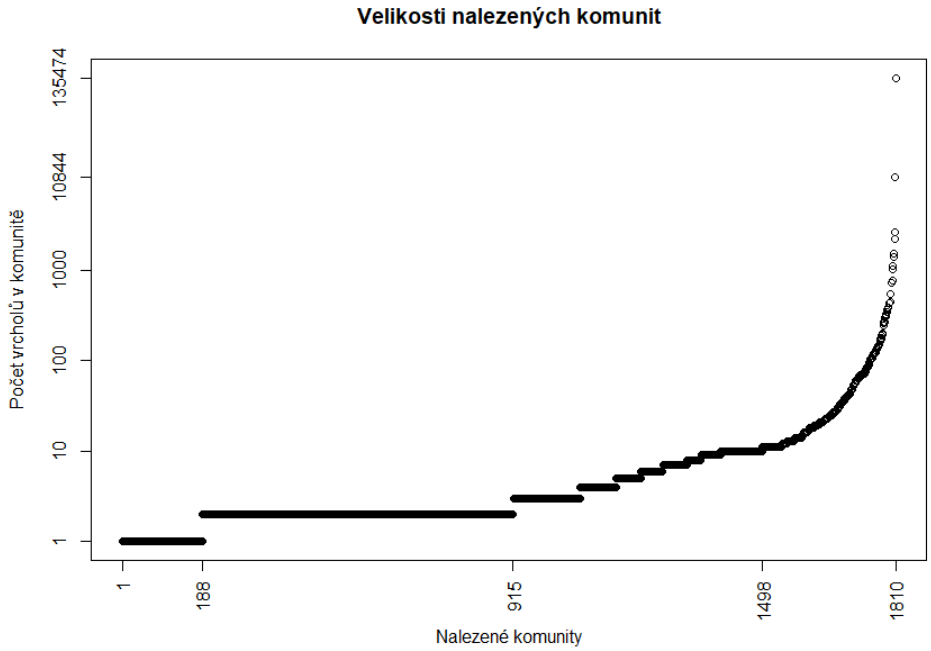
Obrázek 10: Počet změněných labelů v jednotlivých iteracích.

```
def label_propagation(graphData):
    csr: csr_matrix = graphData[0]
    csc = csr.tocsc()
    cols_count = csr.shape[1]
    rows_count = csr.shape[0]
    labels = graphData[1]
    random_order = list(range(cols_count))
    anyChange = True
    iteration = 0
    while anyChange: # Get next generation of labels, until no changes
        random.shuffle(random_order) # Get random order of nodes in each
            generation
        changes = 0
        t = Timer()
        c = Counter()
        for i in random_order:

            # for each node id calculate his new label
            new_label_counter = Counter()

            # load column data
            col = csc.getcol(i)
            edges_weight = col.data
            edges_id = col.indices
            if i < rows_count: # append row data
                row = csr.getrow(i)
                edges_weight = np.concatenate((edges_weight, row.data))
                edges_id = np.concatenate((edges_id, row.indices))

            for id, weight in zip(edges_id, edges_weight):
                target_label = labels[id]
                new_label_counter[target_label] += weight

            # pick random of the most common labels
            new_label = most_common_random(new_label_counter)[0]
            if new_label != labels[i]:
                labels[i] = new_label
                changes += 1

        # write result to file
        np.savetxt(
            f"data/label_propagation_noself/iter_{iteration}.csv", labels, fmt="
                %d")

        # prepare next iteration
        anyChange = changes > 0
        iteration += 1
```

Výpis 2: Implementace LPA v Pythonu.

8 Výsledek LPA

Došli jsme k výsledku algoritmu. Nejlepší představu si uděláme z grafu [11]. Na tomto grafu jsou vykresleny všechny komunity a počet vrcholů, které do komunity patří. Celkový počet komunit, které jsme získali, je 1810 komunit.



Obrázek 11: Nalezené komunity a jejich velikost.

První skupinou komunit jsou takové, které obsahují pouze jeden vrchol. Takových komunit jsme nalezli 188. Vzpomeňme si na kapitolu o komponentách, kde jsme si řekli, že se v grafu nachází 63 vrcholů bez jediné hrany. Takové pak vytvoří také svou vlastní komunitu. Zbytek těchto vrcholů má pravděpodobně více propojení na sebe sama, než na jakýkoli okolní vrchol.

Nejpočetnější skupina jsou takové komunity, které zahrnují dva vrcholy. I to lze vysvětlit pohledem na tabulku slabě propojených komponent [3]. I v té je totiž nejpočetnější skupina komponent o velikost dvou vrcholů z těch můžeme předpokládat vznik stejně velké komunity. Můžeme říct, že tyto komunity pro nás tedy nejsou nijak zajímavé. Podobně málo významné jsou i další malé komunity. Lze říct, že pro nás jsou zajímavé komunity větší než 10 vrcholů. To je sice velká část komunit, 1498 z celkového počtu 1810 představuje 83 %. Protože jsou ale malé, jedná se o relativně malý počet vrcholů. Přesně 5096 vrcholů z celkového počtu 181975. A to je pouze 2.8 %.

Na opačném konci spektra nalezneme jiný extrém. Výsledek obsahuje jednu komunitu, která je větší než všechny ostatní dohromady. Ta zahrnuje 135474 vrcholů, a to je 75 % z počtu 181975

vrcholů. Označil bych tuto komunitu jako "nezařazené", protože ve skupině zahrnující tři čtvrtiny všech grafů bych neočekával přímou souvislost.

Pokud tedy nebudu počítat výše popsané komunity, získali jsme 310 komunit. Doplním ještě, že druhá největší komunita čítá 10844 vrcholů a třetí 2689. Jejich obsah prozkoumám v následující kapitole.

V tuto chvíli se mi algoritmus LPA nejeví jako příliš účinný a jeho výsledky nejsou nikterak oslnivé. Nicméně vybral jsem si ho kvůli jeho rychlosti. A v této disciplíně nakonec vykázal velmi uspokojivý výsledek.

Optimalizace LPA není předmětem mojí práce. Pokud bych se měl přesto zamyslet, jak docílit zkvalitnění výsledku výsledku algoritmu, vyzkoušel bych nějakou optimalizaci. Zajímalo by mě kupříkladu, jaký výsledek by mělo opakované spuštění stejné metody, ovšem omezené jen na největší komunitu z předchozího výsledku. Předpokládám, že by takto vznikly nové, menší komunity. Podobné negativní jevy na kteké jsem narazil, popisuje také článek, který pojednává o upravené variantě nazvané Role-based LPA (roLPA [4]). Řešením by mělo být rozdělení procesu do dvou částí. Nejprve se snaží vyhledat určité jevy v síti a lokalizovat tak centra komunit. Teprve potom se do těchto komunit přidělují ostatní vrcholy. Navíc by takto měl být omezený také problém s oscilací. Zaujal mě také článek o metodě Memory-Based Label Propagation Algorithm (MemLPA [3]). Ten popisuje, že LPA má dva zásadní nedostatky. Že se často uchyluje k lokálnímu optimu, což je řešení, které sice není optimální, ale v daném rozsahu neexistuje žádné lepší. A že často vede ke vzniku jedné gigantické komunity. To přibližné odpovídá našemu problému a autor řeší tento problém přidáním paměti pro jednotlivé vrcholy. Na základě této paměti optimalizuje výběr komunity v dalších krocích.

8.1 Grafické zobrazení komunit pomoci word cloudu

Kdybychom chtěli zkoumat jak úspěšné bylo hledání komunit, zajímalo by nás, zda články zařazené do stejné komunity mají společné téma. Mohli bychom jednotlivé stránky otevřít, přečíst si téma a porovnat ho s ostatními stránkami v komunitě. V případě malých komunit se čtyřmi stránkami by to nejspíš nebyl problém, ale pro větší komunity bychom se tímto způsobem mnoho nedozvěděli. Pro prezentaci jsem tedy zvolil graf, kterému se říká word cloud. V češtině bychom řekli oblak slov. Graf vykresluje vložená slova náhodně do prostoru tak, aby zabírala co největší část prostoru. Slova navíc odlišuje velikostí na základě četnosti výskytu. Větší písmo znamená častější výskyt v textu. Tento graf se často využívá v marketingu. Nám výborně poslouží k tomu, abychom odhadli tématiku komunit. Jako vstup grafu jsem použil titulky stránek.

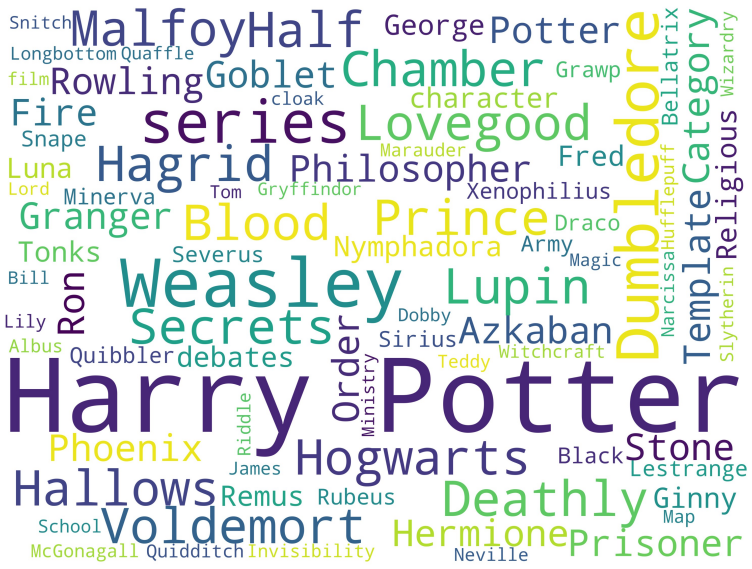
Pro vykreslení jsem využil opět prostředí Python a konkrétně knihovnu Mathplotlib. Zpracoval jsem rovnou všechny komunity. Vygenerované grafy jsem uložil do souborů. Do názvu souboru jsem uložil základní informace s počtem stránek, celkovým počtem slov a nakonec label komunity. Vygenerovanými obrázky jsem tak mohl pohodlně procházet.

Když jsem následně procházel galerii, byl jsem mile překvapen, jak výstižné tyto grafy jsou. Naprostá většina grafů už na první pohled prozrazovala, jakými tématy se zabývají stránky v

této komunitě. Takovými jsou například témata Harry Potter, Pán Prstenů, olympiáda, papežové nebo hudba. Mnoho komunit se vytvořilo okolo geografických nebo geopolitických celků, jako jsou Francie nebo Japonsko. Pro ukázku v této diplomové práci jsem vybral dva grafy, které se mi jevily jako povedené.

Na prvním obrázku [12] vidíme komponentu, kterou bych pojmenoval jako Harry Potter. Tato komunita zahrnuje 57 vrcholů a evidentně se zabývají sedmidílnou sérií knih a filmů od autorky J.K.Rowlingové. V grafu tak rozpoznáme názvy jednotlivých dílů a hlavní postavy celého příběhu.

Druhý příklad vznikl z větší komunity, která zahrnuje 104 vrcholů. Ta zahrnuje různá koření a základní suroviny, které používáme v kuchyni pro dochucování jídla.



Obrázek 12: Word cloud komunity - Harry Potter.

8.2 Visualizace pomocí Gepthi

Jako poslední způsob pro prezentaci výsledků LPA jsem zvolil grafické vykreslení celého grafu. K tomu jsem využil nástroj Gephi. Gephi je rozšířený open-source nástroj pro vizualizaci a zkoumání grafů a sítí.

Už od počátku jsem se ovšem potýkal s technickými problémy. Hned prvním krokem po spuštění programu je otevření datového souboru. Program umí pracovat s formátem GEXF, který je výstupním formátem mého programu. Vybraný vstupní soubor se podařilo otevřít a program jej správně rozpoznal jako orientovaný graf formátu GEXF v1.2. Poté má uživatel

Obrázek 13: Word cloud komunity - koření.

možnost ovlivnit základní parametry grafu. Všechny parametry byly správně rozpoznány, takže jsem potvrdil formulář a začalo vytváření grafu v paměti. Nároky na paměť programu Gephi jsou poměrně vysoké, a tak rychle rozbrazí chybovou hlášku, že vyžaduje více než přidělených 1300MB paměti. Přestože samotná chybová hláška nabízí tlačítko pro navýšení, v mém případě neproběhla žádná změna. Navýšení jsem tedy musel provést v konfiguračním souboru programu, který je na disku součástí instalace. Limit jsem navýšil tak, aby měl rezervu, a nastavil jej na 8GB. Další podmínkou pro využití většího paměťového prostoru je instalace prostředí JAVA v 64-bitové variantě. Také jsem dal na doporučení komunity a vypnul jsem anti-aliasing.

Po těchto úpravách se mi už podařilo graf otevřít a načíst. Po vykreslení grafu, bez dalších manipulací zabíral program 2750MB operační paměti. Taky konstantně využívá 30 % mého procesoru, aniž bych s programem nějak pracoval. Reakce programu byly velmi pomalé a velká část operací končila pádem celého prostředí. Práce s takto velkým grafem tedy byla téměř nemožná. Proto původně vznikl jeden z výše zmiňovaných programů pro sampling metodou Random Walk [5.2].

V kapitolách [7.4.6], kde jsem popisoval práci s Pythonem jsem psal, že jsem největší komponentu grafu uložil do souboru. Konkrétně to byl formát *.gml. Tento typ je také jedním z podporovaných formátu programu Gephi. A narozdíl od původního úplného grafu, tento graf neměl problémy s výkonem. A to především proto, že je výrazně menší. Konkrétně 102.080 vrcholů a 3.032.724 hran. Oproti originálu jsem jej ale rovnou o informaci, do které komunity patří na základě výsledku LPA.

Nyní se můžeme podívat na vyzualizaci grafu, kterou jsem v Gephi vytvořil [14]. Pro rozmístění vrcholů jsem použil layout algoritmus OpenORD [1]. Ten by měl být schopný zpracovat až 1 milión vrcholů. Mohu potvrdit, že dokáže bez problému zpracovat 100 tisíc vrcholů v průběhu méně než pěti minut. Zvýraznil jsem vrcholy tak, aby jejich velikost odpovídala vstupně-výstupnímu stupni. Vhodnější by bylo pro velikost vrcholu využít hodnotu Edge betweenes, ale spočítat tuto hodnotu je výpočetně náročné. Zkusil jsem také vypočítanou metriku HITS. Ta v grafu vyhledává tzv. autority a huby. Ovšem při vykreslení grafu mi tyto hodnoty připadaly méně vypovídající. Ke stejnému závěru jsou dospěl v případě metriky PageRank.

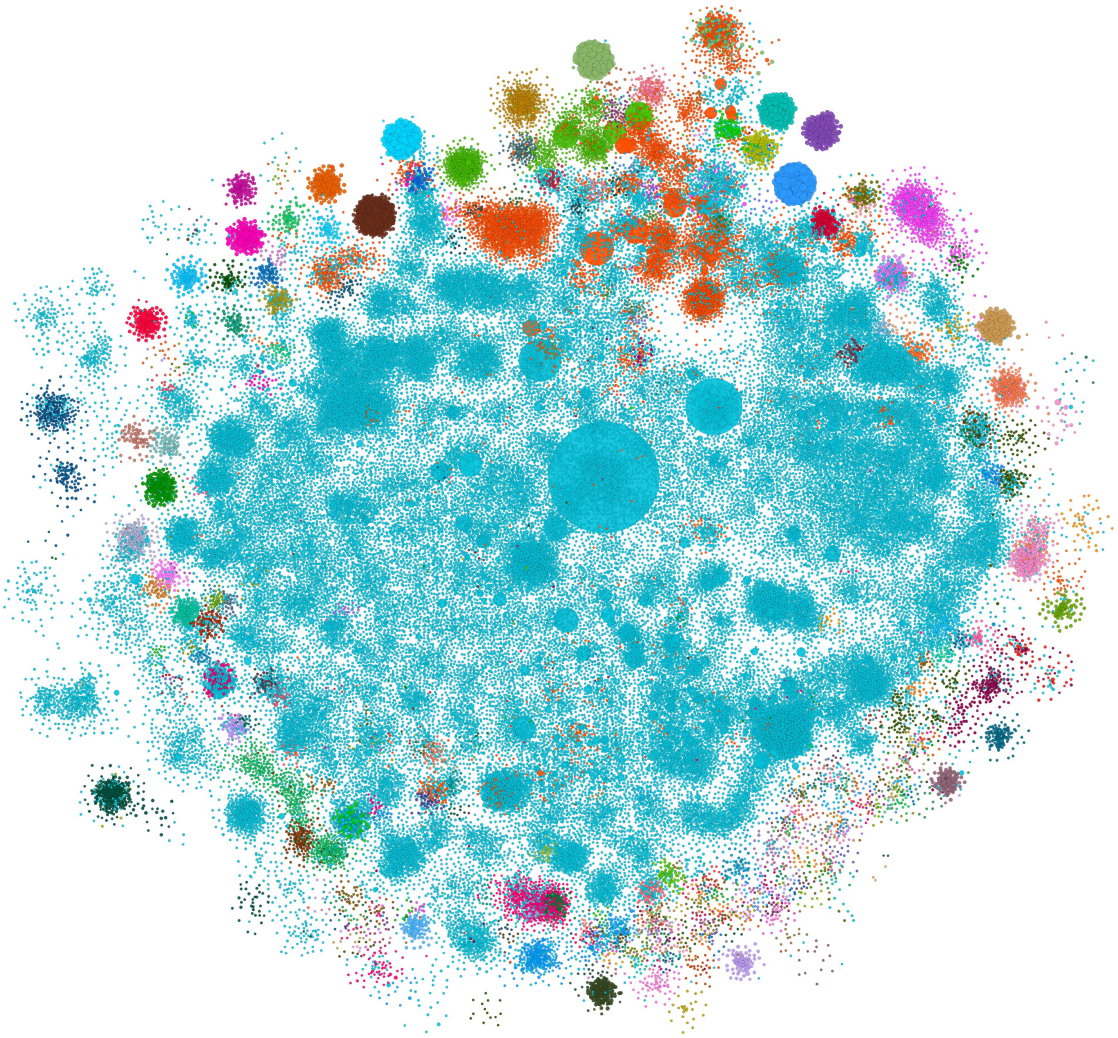
Poté, co jsem graf omezil pouze na největší komponentu, zůstalo v grafu pouze 416 komunit. Každá komunita dostala náhodně přidělenou jednu barvu a touto barvou jsou vykresleny vrcholy, které do komunity patří.

Na obrázku můžeme pozorovat dva výrazné jevy. První je velká skupina světle modrých vrcholů uprostřed grafu, která zahrnuje 75 % vrcholů. Můžeme si všimnout, že zahrnuje také největší vrchol - vrchol s největším stupňem - Stub. Na okraji grafu lze ale nalézt mnoho menších komunit, které jsou výrazně vyčleněné barvou i umístěním. Ty považuji za dobrý výsledek a potvrzuje to i vizualizace word cloud z předchozí kapitoly. Také si můžeme všimnout, že v pravé dolní a dolní části grafu je jakési zrnění, vrcholy různých barev.

Pro porovnání jsem využil nástroj Modularity, který je přímo v programu Gephi. Ten nám nedává takovou volnost, abychom si vybrali algoritmus a implementován je pouze jediný. Můžeme pouze ovlivnit jeho parametry. Detaily tohoto algoritmu můžeme najít v práci Fast unfolding of communities in large networks [**?**]. Ten funguje podobně jako náš LPA a na začátku přiřadí každému vrcholu jednu komunitu. Poté v náhodném pořadí pro každý vrchol hledá tokovou sousední komunitu tak, že když do ní vrchol přesune, zvýší úroveň modularity. Ta se přitom posuzuje spočítáním vah, které vedou uvnitř komunity a mimo ni. Jak popisují autoři této metody, její výsledky jsou velmi dobré. A především algoritmus dosahuje vysokého výkonu, a to do té míry, že graf o velikosti 118 miliónů vrcholů dokázal zpracovat za 152 minut. To ostatně mohu potvrdit, protože v mém případě výpočet trval pouhých pár desítek vteřin.

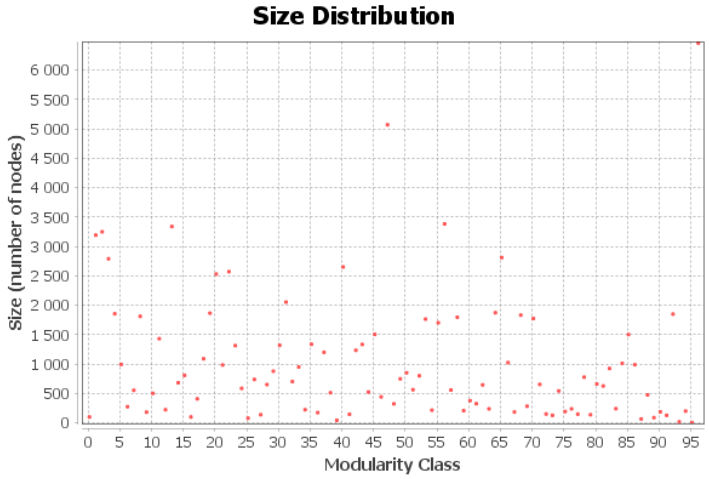
Jak již bylo řečeno, pro tento algoritmus můžeme ovlivnit některé parametry. První dva parametry jsou binární. Jedním si zvolíme, zda má být brána v potaz váha hrany. A druhým si volíme, zda je pořadí vrcholů vyhodnocováno náhodně. To ovšem dle slov autora nemá velký vliv na výsledek. Velký vliv na výsledek má naopak parametr Resolution. Tím do značné míry ovlivníme počet a velikost výsledných komunit. Je třeba s tímto parametrem trochu experimentovat. Vyzkoušel jsem nejdříve jeho výchozí hodnotu 1.0 a následně hodnoty menší a větší. Nejlepšího výsledku jsem dosáhl s hodnotou 0.2.

Výsledek této metody nám dal 97 komunit. Jejich distribuci můžeme vidět na grafu [15]. Z grafu je patrný pěkně rovnoměrný výsledek. Najdeme v něm sice malé množství menších komunit, ale v tomto množství takové v grafu nejspíše opravdu budou. Naproti tomu středně velké komunity o velikosti stovek vrcholů velmi pěkně rozdělují graf na přiměřeně velké části. Dokonce výsledek netrpí na stejný neduh jako moje LPA metoda a neobsahuje žádnou jednu



Obrázek 14: Vykreslený graf největší komponenty zabarvený podle výstupu LPA.

obrovskou komunitu. Největší komunita obsahuje jen 5000 vrcholů.



Obrázek 15: Distribuce velikosti komunit vytvořených nástrojem Gephi.

Tento výsledek jsem následně nechal aplikovat na graf za stejných podmínek jako v předchozím případě. Pouze barvy vrcholů se změnily a odpovídají výsledku metody Fast folding. Při pohledu na obrázek [16] vidíme, že v krajních komunitách byl podobně úspěšný na předchozím obrázku metody LPA. Velmi rozdílně se ovšem jeví celá střední část grafu. Zde se povedlo vrcholy rozdělit do skupin a většina vrcholů tak jednoznačně vytváří nějakou kumunitu. I zde se vyskytuje určitý šum, ale v mnohem menší míře. Navíc je celkový počet komunit mnohem menší (97 oproti 416), a to zejména díky absenci těch nejmenších komunit, které nebyly užitečné.

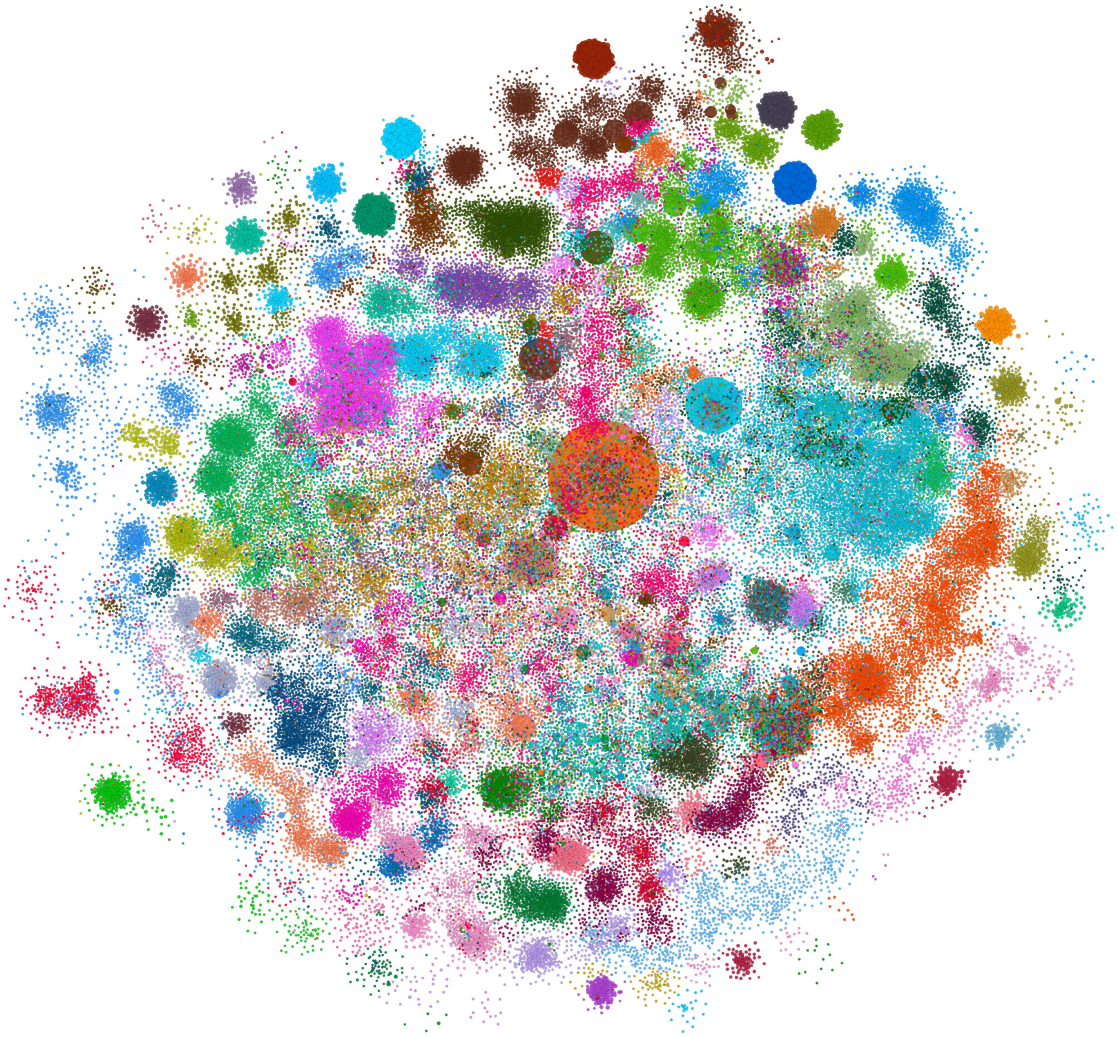
Tento výsledek bych označil za lepší, než ten, kterého jsem dosáhl metodou LPA.

8.3 Vývojový stroj

V tomto dokumentu se často vyjadřuji, jak dlouho daný úkol trval. Aby tento údaj měl nějaký význam, je důležité vědět, na jakém počítači výpočet probíhal. Všechny části vývoje probíhaly na mém stolním počítači s následujícími parametry. [10]

Tabulka 10: Parametry vývojového stroje

Název komponenty	Parametry
Procesor	Intel Core i5-6600K 3.5GHz 4 jádra, 4 vlákna
Operační paměť	16GB, DDR4
Grafická karta	GeForce GTX 970, 4GB
Operační systém	Windows 10 Pro, 64-bit
Internetové připojení	Optické, 130MB/50MB



Obrázek 16: Vykreslený graf největší komponenty zabarvený podle modularity nástroje Gephi.

9 Závěr

Cílem této práce bylo najít vhodnou metodu pro zpracování dat z Wikipedie a pro následnou analýzu získaných dat. Na data jsem nahlížel především jako na síť stránek, které jsou navzájem propojené přes odkazy.

Vývoj probíhal v několika krocích. V každé z nich jsem prozkoumal, jaká jsou možná řešení a vybrané řešení poté použil. Vzniklo několik programů, které postupně z dump souboru získají všechny stránky, převedou je ze syntaxe Wikitext na HTML. Z těch poté vytvoří síť, kterou uloží do formátu GEXF, nebo do komprimované formy tabulky sousednosti ve formátu CSR. Tento postup byl aplikován na databázi Simple English, ale je platný pro libovolnou databázi Wikipedie. Vzniklo také několik dalších programů, které s tímto výstupem pracují. První umožňuje v grafu vyhledávat související články. Druhý dokáže ze souboru načíst graf a vybranou metodou z něj získat vzorek. Tyto programy byly vytvořeny v prostředí .NET Frameworku. V další části jsem se rozhodl, že bude lepší pokračovat v prostředí Python pro jeho časté použití v oblasti computer science a vzhledem k existenci mnoha kvalitních a dobře zdokumentovaných knihoven.

V jazyce Python jsem vytvořil sadu metod, které graf načítá nejprve jako tabulku v matematicko-věděcké knihovně SciPy a dále z ní vytvoří síť v knihovně NetworkX. V této knihovně jsem zdokumentoval nejužitečnější metody, pro výpočet vlastností a metrik grafu. Některé algoritmy měly problém s rozsahem dat. Protože jako jeden z cílů jsem si stanovil vyhledání komunit, zaměřil jsem se na algoritmy řešící tento problém. Protože v knihovně NetworkX jsem nezaznamenal úspěch, vyzkoušel jsem nejprve jinou knihovnu. Po dalším neúspěchu jsem se rozhodl pro vlastní implementaci algoritmu Label Propagation (LPA). Na vlastním řešení jsem dokázal pochopit důvod, proč předchozí pokusy nebyly úspěšné. Vyřešením problému jsem úspěšně roztřídil vrcholy do komunit. Výsledek jsem dále prezentoval v grafické podobě pomocí word cloud a vykreslením celého grafu s barevným rozlišením komunit. Pro vykreslení jsem použil program Gephi. Ten zároveň implementuje vlastní metodu hledání komunit, kterou jsem srovnal s metodou LPA. Výsledek LPA z tohoto porovnání vyšel jako méně kvalitní.

Přesto, že množství dat ani velikost grafu není až tak vysoká, od počátku jsem mnohokrát musel řešit výkonnostní limity a často kvůli nim řešení optimalizovat nebo zvolit jiné. Nejen že některé banální operace trvaly často několik minut, složitější operace i několik dní, ale některé algoritmy by počítaly donekonečna. Další komplikace plynoucí z velkého množství stránek je problematické hledání chyb. Výsledky jednotlivých kroků si lze ručně zkontrolovat, ovšem pouze v omezeném rozsahu na několika příkladech. Protože vstupem každého kroku je výstup některého z předchozích, případné chyby se tak často projevily až v některém z následujících kroků. Při jejich opravě je tak nutné od chybné části proces opakovat a všechny další výsledky přepočítat, což zvýšilo časovou náročnost projektu.

Podařilo se mi naplnit všechny stanovené cíle. Vzniklo několik programů, nástrojů a skriptů, které lze použít pro získání a analýzu dat. Dokument popisuje nejen, jak tento proces funguje,

ale především jaké jsou důvody pro jednotlivá řešení. Může tedy posloužit jako vodítko pro podobné projekty. Zároveň jsou ale zajímavé také výsledky analýzy sítě.

Literatura

[1] M. Bastian. Openord, Dec 2018.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. 2008.

[3] A. Fiscarelli, M. Brust, G. Danoy, and P. Bouvry. *A Memory-Based Label Propagation Algorithm for Community Detection*, pages 171–182. 12 2018.

[4] X. Hu, W. He, H. Li, and J. Pan. Role-based label propagation algorithm for community detection, 2016.

[5] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[6] NetworkX developer team. Networkx, 2014.

[7] T. P. Peixoto. Graph-tool performance comparison, Oct 2015.

[8] Zhang, Aiping, Ren, Guang, Lin, Yejin, Jia, Baozhu, Cao, Hui, and et al. Detecting community structures in networks by label propagation with prediction of percolation transition, Jul 2014.

A Obsah digitální přílohy

- R.zip
  - R Lang skript pro výpočet statistických hodnot a vykreslení grafů.
- PythonNetworkAnalysis.zip
  - Všechny Python skripty, které byly v této práci použity.
- largest_subgraph_component_titles_LPA.zip
  - Největší komponenta grafu ve formátu GML. Vrcholy zahrnují atributy ID, Title a výsledek LPA.
- sparseMatrisCSR_titles_LPA.zip
  - index.titles.txt - seznam všech vrcholů, jejich ID a Titulek
  - LPA_result.csv - Výsledek algoritmu LPA
  - SparseMatrixCSR.csv - Tabulka sousednosti uložená ve formátu CSR
- top100_nodes_edges.zip
  - graph_top100Edges.csv - Prodloužený výpis 100 hran s nejvyšší váhou
  - graph_top100Nodes.csv - Prodloužený výpis 100 vrcholů s nejvyšším stupněm
- WikiParser.zip
  - Kompletní adresář Visual Studio Solution, který obsahuje všechny projekty vytvořené ve Visual Studiu.
- wordcloud.zip
  - Výběr 6-ti obrázků s word cloud grafem. Každý vyobrazuje jednu komunitu.